COMPACT CONNECTIVITY REPRESENTATION FOR TRIANGLE MESHES

A Thesis
Presented to
The Academic Faculty

by

Topraj Gurung

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Interactive Computing

Georgia Institute of Technology
May 2013

COMPACT CONNECTIVITY REPRESENTATION FOR TRIANGLE MESHES

Approved by:

Professor Jarek Rossignac, Advisor
School of Interactive Computing
Georgia Institute of Technology

Professor David Frost, Co-advisor
School of Civil and Environmental Engineering
Georgia Institute of Technology

Dr. Peter Lindstrom
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory

Professor Greg Turk
School of Interactive Computing
Georgia Institute of Technology

Professor Karen Liu
School of Interactive Computing
Georgia Institute of Technology

Date Approved: Jan 10 2013

To my parents,

for their love and support,

and for instilling the strength and self-belief

to pursue my dreams.

ACKNOWLEDGEMENTS

I want to first thank my advisor Jarek Rossignac. Jarek's brilliance and insights continue to amaze me, and I feel fortunate to have been advised by him. I am thankful for the many conversations we had and advice he gave me on research and life in general. He is truly a wonderful mentor.

I would like to thank my advisor David Frost. I worked with DF the whole time I was at Georgia Tech since my undergraduate days through my Master's and finally my PhD. I am forever grateful to him for encouraging me to pursue a PhD. I always looked forward to the conversations we had over cups of latte discussing life and a broader vision on universities and research. His vision is inspiring and continues to drive me. It has been an honor to learn from him, and I hope our collaboration will continue for years to come.

I would like to thank my collaborator Peter Lindstrom who has been an unofficial advisor to me. My collaboration with Peter started when he hosted me for a summer internship at LLNL in 2010, and it has been a wonderful collaboration since then. I have admired his insights, his patience and attention to detail. I feel blessed to have had three advisors mentor me during my PhD.

I would like to thank Greg Turk and Karen Liu for serving in my committee, and for their continual help and advice during my PhD.

I would like to thank the students from the Georgia Tech graphics group, past students Brian Whited, Justin Jang, Jason Williams, Yuting Ye, and Sumit Jain and present students Mark Luffel, Tina Zhuo, Jie Tan, Karthik Raveendran, Kristin Siu, Yunfie Bai, Sehoon Ha and many others. A special thanks to Mark who has been a collaborator and a good friend, and also to fellow graduate students in the MAGIC

lab. You have all made grad school an incredibly rewarding experience.

And finally, I would like to thank my grandparents, uncles, aunts and cousins for their continual encouragement and support. Even though they are scattered all across the world in Nepal, UK, USA and Hong Kong, their support was always there. And most importantly, I would like to thank my mother, my father and my brother for their love and support, and their belief in me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Many digital models used in entertainment, medical visualization, material science, architecture, Geographic Information Systems (GIS), and mechanical Computer-Aided Design (CAD) are defined in terms of their boundaries. These boundaries are often approximated using triangle meshes. The complexity of models, which can be measured by triangle count, increases rapidly with the precision of scanning technologies and with the need for higher resolution. An increase in mesh complexity results in an increase of storage requirement, which in turn increases the frequency of disk access or cache misses during mesh processing, and hence decreases performance. For example, in a test application involving a mesh with 55 million triangles in a machine with 4GB of memory versus a machine with 1GB of memory, performance decreases by a factor of about 6000 because of memory thrashing. To help reduce memory thrashing, we focus on decreasing the average storage requirement per triangle measured in 32-bit integer references per triangle (*rpt*).

This thesis covers compact connectivity representation for triangle meshes and discusses four data structures:

1. Sorted Opposite Table (SOT), which uses 3 *rpt* and has been extended to support tetrahedral meshes.

2. Sorted Quad (SQuad), which uses about 2 *rpt* and has been extended to support streaming.

3. Laced Ring (LR), which uses about 1 *rpt* and offers an excellent compromise between storage compactness and performance of mesh traversal operators.

4. Zipper, an extension of LR, which uses about 6 bits per triangle (equivalently

0.19 *rpt*), therefore is the most compact representation.

The triangle mesh data structures proposed in this thesis support the standard set of mesh connectivity operators introduced by the previously proposed Corner Table [35] at an amortized constant time complexity. They can be constructed in linear time and space from the Corner Table or any equivalent representation. If geometry is stored as 16-bit coordinates, using Zipper instead of the Corner Table increases the size of the mesh that can be stored in core memory by a factor of about 8.

CHAPTER I

INTRODUCTION

A fraction of digital models used in entertainment, medical visualization, architecture, GIS, and mechanical CAD are defined in terms of their boundaries. These boundaries are often approximated using triangle meshes. The complexity of models, which can be measured by triangle count, increases rapidly with the precision of scanning technologies and with the needs for higher resolution. An increase in mesh complexity results in an increase of storage requirement, which in turn increases the frequency of disk access or cache misses during mesh processing, and hence decreases performance. For example, in a test application involving a mesh with 55 million triangles in a machine with 4GB of memory versus a machine with 1GB of memory, performance decreases by a factor of about 6000 because of memory thrashing.

To help reduce memory thrashing, we focus on decreasing the average storage requirement per triangle measured in 32-bit integer references per triangle (*rpt*).

Although we focus on triangle meshes, we would like to point out that there are other ways to represent surfaces and solids. Solids can be represented by voxels from which an iso-surface can be extracted that approximates the bounding surface. Alternatively, solids can be represented as a combination of primitives using Constructive Solid Geometry (CSG). Surfaces can be represented as parametric surfaces defined by a control mesh. In our dissertation, we focus on triangle meshes because they are the simplest and most popular representation for surface approximations. Triangle meshes have become the common denominator for representing surfaces for many applications. Some process meshes directly – most can import/export them. Smooth surfaces produced by iso-surface extraction, trimmed by CSG evaluation, or

defined in terms of a control mesh can be converted to an approximating tessellation represented using a triangle mesh. Finally, there is abundant hardware support for accelerated processing and rendering triangle meshes.

1.1 Triangle Mesh representation

To simplify our discussion of the storage required by a particular mesh representation, we distinguish the geometry from the connectivity.

1.1.1 Geometry

The geometry of a mesh refers to the location of points (called vertices) that are sampled to approximate the surface of the model. The vertices are often each represented by their x, y, z coordinates.

1.1.2 Connectivity

Early representations of connectivity graphs [5] use 32-bit pointers. Some of the recent representations (for example [35]) assign consecutive non-negative integers to vertices and triangles and use arrays to store sorted lists of references (integer indices), rather than pointers. In our work, we assume that the mesh is manifold and also assume that the vertices and triangles have non-negative consecutive IDs, and we access the vertices and triangles using the integer indices. Connectivity information may be decomposed as follows:

1. Incidence:
   (a) Triangle-vertex incidence: Each triangle is the convex hull of its three vertices. This incidence relation defines the three vertices.
   (b) Vertex-triangle incidence (star): Each vertex has incident triangles.
2. Adjacency:

(a) Triangle-triangle adjacency: Each triangle shares an edge with up to three neighboring triangles in a manifold mesh.

(b) Vertex-vertex adjacency: Two vertices that share an edge are adjacent to each other.

3. Ordering:

(a) Vertices around triangles: The vertices of a triangle can be listed in a consistent orientation (see Section 2.1).

(b) Triangles around vertices: The triangles incident on a vertex can be listed in a consistent orientation (see Section 2.1).

When the mesh is an orientable manifold, the three kinds of connectivity information listed above can be inferred algorithmically from the triangle-vertex incidence.

Hence, the popular Indexed Mesh (incidence) format [8], which specifies the indices of the three vertices of each triangle, suffices to describe the mesh without ambiguity.

Examples of such format include ply, stl and obj which are used in graphics and animation and can be exported from most commercial design software such as Maya [4], AutoCAD [3] and 3dsMax [2].

1.2 Connectivity operators

The information stored in the Indexed Mesh format alone is not sufficient for practical applications because access to an adjacent triangle or to a neighboring vertex requires, on average, a search through a large fraction of the mesh. So, to support random-access and mesh traversal operators (discussed below) in average constant time, most representations store additional information, which often corresponds to a more complete connectivity graph including several of the link types discussed above in Section 1.1.2. To traverse these connectivity graphs, applications manipulate operators that map tuples of entities ⟨vertex, edge, triangles⟩ to other tuples.

In particular, many mesh processing algorithms may be formulated using operations that access the next vertex around a triangle or the next triangle around a vertex. Hence, we use a corner [35] (a vertex-triangle tuple identifying a triangle and one of its vertices) as a primitive (iterator) for mesh traversal.

An extended and more convenient set of corner operators useful to application developers may be trivially derived from a small set of core operators. There are several possibilities for selecting the core set. We propose the following *core set* (see Fig. 1), where c defines a corner.

- the triangle $c.t$ containing c,
- the vertex $c.v$ of c,
- the next corner $c.n$ in $c.t$,
- the opposite corner $c.o$ defined as the corner o such that $o.n.v == c.n.n.v$ && $c.n.v == o.n.n.v$,
- the first corner $t.c_0$ of triangle t, which is the corner with the smallest ID of triangle t, and
- an arbitrary corner $v.c$ of vertex v.

From these, we derive an extended set of convenient operators as shown below, which we call the *derived set* (see Fig. 1):

- the previous corner $c.p = c.n.n$ in $c.t$,
- the left $c.l = c.n.o$ and
- the right $c.r = c.p.o$ neighboring corners of c, and
- the swing $c.s = c.l.n$ and
- the unswing $c.u = c.r.p$ corners used to walk around $c.v$.

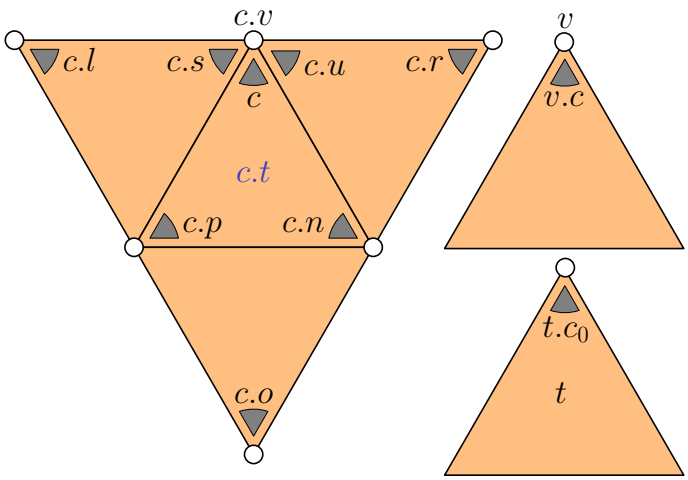


Figure 1: The standard set of corner operators. For a corner c, a vertex v, and a triangle t, the core set consists of the triangle $c.t$, vertex $c.v$, next corner $c.n$, opposite corner $c.o$, first corner $t.c_0$ and an arbitrary corner $v.c$. The *derived set* consists of the previous corner $c.p$, left corner $c.l$, right corner $c.r$, swing corner $c.s$ and unswing corner $c.u$.

The *standard set* of corner operators is the union of the core set and the derived set. We define the action of *swinging* around a vertex v as repeatedly calling the swing $c.s$ operator, so as to visit all the corners incident on the vertex v.

1.3 Randomly Accessible and Traversable (RAT) compatible representations

The representations presented in this thesis are RAT compatible. A mesh representation is a RAT compatible representation if it provides *constant-time* (amortized) and *fast* retrieval of any triangle, vertex, or corner (equivalently half-edge) by their ID, and also supports constant-time (amortized) access to neighboring (adjacent, incident) elements in a consistent order, e.g. the consecutive vertices around a triangle and the consecutive incident triangles around a vertex.

1.4 Reduction of storage cost for connectivity

In this thesis, we focus on reducing storage cost for connectivity. This is important because in most representations, the storage cost for connectivity exceeds the storage

cost for geometry (vertex locations). To help us justify this claim and to simplify comparison, we use the average number of *references per triangle* (abbreviated *rpt*) as the measure of storage. The references are 32-bit integer indices that identify vertices, corners, or other entries in the arrays. If a representation uses rn references where n is the number of triangles, we say that its storage cost is r *rpt*.

1.4.1 Domain

The set of meshes targeted in this thesis is limited to manifold triangle meshes possibly with a boundary. The storage claims reported throughout the thesis are valid for cases where the mesh has low genus and relatively few boundary edges (edges that have only one incident triangle) compared to the number of triangles.

Typically, storage cost has a component proportional to the number of vertices and another component proportional to the number of triangles. Hence, to compare two different storage costs, we need to express the per vertex cost and per triangle cost in a common currency. To do so, we use the fact that under certain assumptions about the mesh topology, there are roughly twice as many triangles as vertices in a mesh.

Specifically, if the mesh is watertight (closed and orientable), the Euler-Poincare characteristic χ is given as:

$$\chi = V - E + F = 2 - 2g$$

where V is the number of vertices, E is the number of edges, F is the number of faces, and g is the genus of the mesh.

In a closed watertight triangle mesh, since each edge is shared by two faces and each face has 3 edges, we get:

$$E = (3/2)F$$

Hence, the Euler-Poincare characteristic becomes:

$$n = 2m - 4 + 4g$$

where n is the number of triangles, m is the number of vertices and g is the genus. Assuming the number of boundary edges and genus is negligible with respect to n, there are roughly $n/2$ vertices and $3n/2$ edges.

1.4.2 Storage cost for geometry

To understand the relative importance of storage cost of connectivity, let us look first at the storage cost of geometry. Often, developers store coordinates of vertices as a 32-bit floating-point number (which yields a cost of 3 reference per vertex which is equivalent to 1.5 *rpt*).

However, in most applications, vertex coordinates may be quantized without jeopardizing accuracy requirements to reduce the storage cost of geometry [17]. For example, when a model is used in a digital mock-up to approximate a car engine, a 16-bit quantization of each coordinate provides about 0.02 mm accuracy, which is sufficient for most applications. Hence, storage of quantized geometry amounts to 0.75 *rpt*.

1.4.3 Storage cost for connectivity

Connectivity can be stored using various representation schemes. Here, let us briefly review the storage cost of two examples of connectivity representations.

First, we consider a simple toy connectivity representation which explicitly stores most of the connectivity information. Specifically, storing the six core corner operators (see Section 1.2) as look-up tables would require 13.5 *rpt*. Let us explain this conclusion. In this scheme, for every corner, the result of the $c.t$, $c.v$, $c.n$ and $c.o$ operators are stored. Since there are 3 corners per triangle, storing the result of these 4 corner operators requires 12 *rpt*. Additionally, in this scheme, for every triangle, the result of the $t.c_0$ operator is stored. This results in storing 1 *rpt*. Finally, in this scheme, for every vertex, the result of the $v.c$ operator is stored. Since there are

roughly half the number of vertices as the number of triangles, storing the result of the $v.c$ operator requires 0.5 *rpt*. Summing up 12 *rpt*, 1 *rpt* and 0.5 *rpt* results in a total storage requirement of 13.5 *rpt*.

The representation discussed above does not assume any particular order of the vertices, corners or triangles. To reduce connectivity storage, some representations reorder vertices, corners, or triangles. For example, ECT (the Extended Corner Table) [25, 35] encodes connectivity using 6.5 *rpt*, by assigning consecutive integers to the three corners of each triangle. In this scheme, for each corner, the results of the $c.v$ and $c.o$ operators are stored. Additionally, for each vertex, the results of the $v.c$ operator is stored. Since there are 3 corners per triangle and there are roughly half the number of vertices as the number of triangles, the storage cost is 6.5 *rpt*. Assigning consecutive integers to the three corners of each triangle in a consistent order makes it unnecessary to store several of the references stored explicitly in the scheme discussed previously, because their content may be computed in constant time when needed: for example, triangle $c.t = \lfloor c/3 \rfloor$, next corner $c.n = 3{*}c.t + (c+1)\%3$, previous corner $c.p = 3{*}c.t + (c+2)\%3$ and the first corner $t.c_0 = 3 * t$, where % denotes the modulo operation.

Although the second scheme is twice more compact than the first one, connectivity in the second scheme still dominates total storage. As the connectivity storage cost is 6.5 *rpt* and geometry, if stored as 16-bit coordinates, has storage cost of 0.75 *rpt*, the total storage cost is 7.25 *rpt*. Therefore, connectivity accounts for about 90% of the total storage. Hence, it is important that we devise more compact representations of connectivity. It should be noted that nontrivial applications might store auxiliary data along with vertices and triangles, e.g. flags, colors, normals, in which case connectivity accounts for a smaller percentage of the total storage cost.

1.5 Performance

Besides our goal of reducing storage for connectivity, our other goal is to provide a data structure for which the corner operators are fast. By fast, we mean that the running time of the corner operators has constant time complexity. More precisely, we accept solutions where the running time has an average constant time complexity, meaning that if we apply the same operator to all entities, the average would be constant. For example, the running time of the computation of the valence of a given vertex v, by swinging around v, is a function of the valence (number of incident triangles) of v. Yet, we say that computing the valence has an average constant time complexity because when executed for all vertices, the average cost is constant. The reason why the average is constant is because the average number of corners per vertex, i.e. the valence is 6 for a manifold mesh with low genus and no boundary.

To help understand how our data structures perform relative to the state-of-the art, we report comparison of timing results for various micro-benchmarks.

1.6 Alternative connectivity representations

To appreciate the data structures proposed in this dissertation, we briefly discuss below alternative representations where either too much or too little information is stored.

Now, let us explore a connectivity representation that stores the results of all the core and derived corner operators (Section 1.2) in *lookup tables*. With this scheme, each connectivity query (Section 1.1.2) can be answered with one lookup. In this scheme, for every corner, the results of the nine corner operators ($c.t$, $c.v$, $c.n$, $c.o$, $c.p$, $c.l$, $c.r$, $c.s$ and $c.u$) are stored. Since there are 3 corners per triangle, storing these nine corner operators requires 27 *rpt*. Additionally, for each triangle, the result of the $t.c_0$ operator is stored. This requires an additional 1 *rpt*. Finally, for each vertex, the result of the $v.c$ operator is stored. As there are roughly half the number

of vertices as number of triangles, storing the result of the $v.c$ operator requires 0.5 *rpt*. Summing 27 *rpt*, 1 *rpt* and 0.5 *rpt* results in a total connectivity storage cost of 28.5 *rpt*. As an aside, as noted in Section 1.2, the derived set of corner operators can be expressed as a combination of the core ones. Hence, it is not necessary to store the lookup tables associated with the derived corner operators.

Conversely, if an insufficient amount of information is stored, it may not be possible to answer some connectivity queries in average constant time. We now explore this problem. Consider a data structure that stores only the triangle-vertex incidence lookup table [35]. In this scheme, triangle entries are stored as triplets of vertices with a consistent ordering. This scheme requires 3 *rpt*. The $c.v$, $c.n$ and $c.p$ operators take constant time. But other corner operators, such as $v.c$ and $c.o$, in the worst case scenario, requires scanning the entire vertex table. Hence, these operators have linear complexity in the number of corners in the vertex table.

Let us look at a more extreme example that leads to an even more compact representation: a compressed format. For example, if a compressed representation, such as Edgebreaker [35] or its variation [19], is used, the storage requirement is guaranteed to not exceed 1.80 bits per triangle (*bpt*). However, before connectivity queries can be answered, the compressed representation must be decompressed. The complexity of the decompression algorithm is linear in the number of triangles. Hence, a connectivity query that starts by decompressing the model has a linear time complexity in the number of triangles. Furthermore, the decompression algorithm requires additional storage for the decompressed model. It should be noted that storage of such a non-compressed data structure is the topic of this thesis.

1.7 Contribution

The data structures that are proposed in this thesis support the complete set of core and derived corner operators in constant time, or at least in constant average time.

The performance of these operators is better or comparable to the performance of operators supported by previously proposed approaches.

In this dissertation, we present four data structures for triangle meshes:

1. Sorted Opposite Table (SOT) [25], which uses about 3 *rpt* and has been extended to tetrahedral meshes [24],

2. Sorted Quad (SQuad) [21], which uses about 2 *rpt* and has been extended to streaming [30],

3. Laced Ring (LR) [22], which uses about 1 *rpt* and supports an efficient implementation of the corner operators,

4. Zipper [23], an extension of LR, which uses about 6 *bpt* (equivalently 0.19 *rpt*), but for which the operators are not as efficient as those for LR

The storage result achieved in Zipper is remarkable because it improves by 5.8x the result (LR) of a long-standing research thread that has already managed, over 40 years, to reduce connectivity storage by 12x from 432 *bpt* [6] to 35 *bpt* [22].

As per our requirement, Zipper provides support for all the standard corner operators in constant time. Furthermore, Zipper supports all standard corner operators, yet instead of the lookup tables discussed above, it stores only about one fifth of a 32-bit reference. As such, this result approaches the compactness of compressed connectivity formats.

1.7.1 Principles

In this thesis, we explore how global or local reorderings of vertices, corners and triangles can be exploited to provide implicitly the information that was stored in the lookup tables discussed above.

We have previously discussed one particular reordering. In the Extended Corner Table, the corner-vertex lookup information is ordered so as to implicitly provide the

$c.n$ and $c.p$ corner operators. We do so by ensuring that the three vertices are stored consecutively as triplets in an array. As a consequence, we can compute the $c.n$ and $c.p$ operators in constant time (see Section 1.4.2).

The contributions reported in this thesis are based on combining this corner reordering with specific reordering of triangles, vertices or both.

The *first principle* that we propose is synchronizing the vertex and triangle numbering. The objective is to match triangles and vertices so that the ID of one implicitly defines the ID of the other.

However, remember that there are roughly twice as many triangles as vertices. Therefore, the *second principle* that we propose is to group two adjacent triangles with one of their shared vertices. For the elements for which such a match is established, the ID of the triangle or of a vertex implicitly defines the ID of the other elements of the group. An additional benefit of such grouping is that the $c.o$ corner operators between the opposite corners of the two triangles of a group need not be stored because they can be inferred from the relative indices of the corners.

The *third principle* that we propose is to renumber the vertices along a loop which we call a ring so as to chain the groups. A benefit of such chaining is that the vertex IDs of the vertices that bound the edge shared between the two triangles in a group need not be stored because the IDs are consecutive along the ring.

The *fourth principle* that we propose is to store differences between consecutive entries in an array instead of the individual integers. This works well when the difference fits in a few bits, therefore instead of storing a whole 32-bit integer, we can store a smaller integer to reduce the storage cost.

1.7.2 Summary of the four approaches

SOT [25] matches each vertex with a different triangle and reorders triangles so that triangle i of the first m triangles corresponds to vertex i, where m is the number

of vertices. Hence, there is no need to store the incidence V table, which may be recovered by swinging around each vertex until a triangle is reached with a sufficiently small identifier. The $v.c$ operators is also available implicitly. Hence SOT uses 3 *rpt*.

SQuad [21] extends SOT by pairing most unmatched triangles with matched vertex-triangle combination to form matched quads. By forming quads, SQuad avoids storing one opposite corner per triangle between triangles in the same quad. Hence SQuad uses slightly more than 2 *rpt*.

LR [22] reorders the vertices and matched incident quads of a mesh along a nearly Hamiltonian cycle called the ring. It associates the two triangles incident on a (directed) ring edge e with the vertex v at the source of the edge. LR then stores, for each v, the (integer) references $v.L$ and $v.R$ to the tip vertices (those not on e) of the two triangles that these vertices form with e. Given that most vertices are in the ring, this amounts to storing one reference (32 bit) per triangle. In LR, many adjacency relationships can be inferred from the ring. LR stores on average about 1 *rpt*.

Zipper extends LR and avoids storing most of the $v.L$ and $v.R$ references explicitly. Instead, it stores a pair of 3-bit codes for most ring vertices. These codes store delta increments between two $v.L$ ($v.R$) consecutive entries from which $v.L$ ($v.R$) are derived in constant time. Zipper stores on average about 6 *bpt* (equivalently 0.19 *rpt*).

CHAPTER II

BACKGROUND

To provide the reader with background material on the thesis, in this chapter, we briefly summarize material relevant to triangle meshes and their representations.

2.1 Simplicial complexes and various properties

A surface can be decomposed into vertices, edges and triangles, which we refer to as cells. We say that a cell b bounds a cell c if b is part of the boundary of c, relative to the closure of that manifold. The *star* of a cell is the union of the cells it bounds.

Restricting the cells to be the convex hulls of their vertices and assuming that all cells are contained in the shape yields a simplicial complex, which decomposes a shape into relatively-open cells: 0-cells are the vertices, 1-cells are the edges excluding their *bounding* vertices and 2-cells are the triangles excluding their bounding edges and vertices. We say that an edge is *incident* upon its bounding vertices, and a triangle upon its bounding vertices and edges. Two triangles are *adjacent* if they are incident upon the same edge, and two vertices are adjacent if they are bounding the same edge.

Simplicial complexes have a regular representation: a vertex may be represented by 1 vertex reference, an edge by 2 vertex references and a face by 3 vertex references. More generally, a k-cell is defined by $k+1$ vertex references.

The k-graph of a simplicial complex has nodes each representing a different k-cell and has a link between such two nodes when the corresponding k-cells are adjacent.

We say that the simplicial complex is a k-mesh, when it satisfies the following conditions:

1. No self-intersection (proper imbedding): the intersection of any two different cells is empty (remember that we define cells as relatively open)

2. No dangling cells (dimensional homogeneity): each m-cell with $m < k$ is in the boundary of at least one k-cell,

3. Manifold: the star of every m-cell with $m < k$ is connected.

4. Connected: the union of all the cells is connected.

5. Orientabile: the orientation of all k-cells is compatible across shared boundaries.

A triangle mesh is a 2-mesh. The conditions above do not support an explicit representation of non-manifold cases. A triangle mesh that is non-manifold and that represents the boundary of a solid may be converted to pseudo-manifolds while minimizing vertex replication [36].

In a triangle mesh, if there is only one triangle incident on a bounding edge, the edge is a *border edge*. The two vertices bounding the border edge are called *border vertices*. A non-border vertex is called an *interior vertex*.

We assume that the vertices are numbered from 0 to $m-1$, where m is the number of vertices, and that the vertex locations are stored in a geometry table G, where $G[v]$ is the point where vertex number v is located. Other vertex attributes (density, color, normal) may also be stored, but are not discussed here.

In a triangle mesh, the incidence information is typically stored as a 3-tuple of triangle-to-bounding-vertex references (integer indices) that identify the bounding vertices of a triangle [9]. Selecting an order for listing these references defines one of two possible orientations of the triangle.When the triangle mesh represents the boundary of a solid s, the order (A,B,C) in which the vertex references are listed in is chosen so that the vector $AB \times AC$, when placed at $(A+B+C)/3$ points outwards of s. More generally, a triangle mesh is *oriented* when two triangles t_1 and t_2 are

incident upon vertices A and B, where t_1 is defined by vertices (A, B, C) and t_2 by (A, B, D) then t_2 must have the one of the following 3 sequences of references (B,A,D), (D,B,A), or (A,D,B). We assume that the mesh is *orientable* (that a consistent orientation exists, for example, a Mobius strip is not orientable) and that the order of vertex references reflects this orientation.

A doughnut or a coffee mug has a through hole, which we refer to as the *handle*. We can compute the *genus*, which is the number of handles, of a mesh by using the Euler-Poincare formula. For a manifold orientable triangle mesh with no border edges, as was noted in Section 1.4.1, we have:

$$2m = n + 4 - 4g$$

where m is the number of vertices, n is the number of triangles and g is the genus.

A triangle mesh is *watertight* if it does not contain any border edges or vertices, and is orientable.

The *valence* of a vertex is the number of triangles incident on the vertex. The average valence of a watertight triangle mesh with a relatively high number of vertices and low genus is close to 6.

2.2 Corner operators

The data structures proposed in this dissertation access elements (corners, vertices, triangles) of the triangle mesh by using corner operators. A corner [35] (a v-t tuple identifying a triangle and one of its vertices) is a primitive (iterator) for mesh traversal. We discussed the corner operators in Section 1.2. In that section, we used the opposite corner $c.o$ as a core corner operator. Alternatively, we can define the swing corner $c.s$ as a core corner operator.

- the swing corner $c.s$ is defined as the corner s such that $s.v == c.v$ && $s.n.v == c.p.v$,

If we had defined $c.s$ as a core corner operator instead of the $c.o$ operator, the derived set would be defined as:

- the previous corner $c.p = c.n.n$ in $c.t$,
- the left $c.l = c.s.p$ and
- the right $c.r = c.n.s.p$ neighboring corners of c, and
- the opposite $c.o = c.p.s.p$ and
- the unswing $c.u = c.n.s.n$ corners used to walk around $c.v$.

2.2.1 Examples of corner operator usage

To demonstrate the use of corner operators, we list the following example operations for a triangle mesh.

2.2.1.1 Computing valence

We can compute the valence of a non-boundary vertex v by using the following pseudocode:

```
valence = 0;
sc = v.c;
c = sc;
do {
    c = c.s;
    valence++;
} while(c!=sc);
```

2.2.1.2 Depth First Traversal

We can visit all the triangles of a mesh by performing a depth first traversal of a Triangle Spanning Tree (dual graph). Starting from a triangle t, use the following pseudocode:

```
sc = t.c0;
mark(t);
stack.push(sc);
while(stack not empty) {
    c = stack.pop();
    if(unmarked(c.l.t)) {
        mark(c.l.t);
        stack.push(c.l);
    }
    if(unmarked(c.r.t)) {
        mark(c.r.t);
        stack.push(c.r);
    }
    if(unmarked(c.o.t)) {
        mark(c.o.t));
        stack.push(c.o);
    }
}
```

2.3 Corner Table

The Corner Table (CT) promoted by Rossignac et al. [35] provides a simple and efficient representation of triangle meshes, storing 6 integer references per triangle (3 vertex references in the Vertex table V and 3 references to opposite corners in the Opposite table O).

For each corner c of each triangle, the Corner Table stores the references $V[c]$ to the corresponding vertex and the reference $O[c]$ to the opposite corners in an adjacent triangle (as shown in Fig. 2), if one exists. It does not store any references from vertices to corners or to incident triangles. The V Table lists the triangle-vertex incidence, such that the 3 vertices bounding a triangle t are consecutive ($V[3t]$, $V[3t+1]$, $V[3t+2]$) and listed in an order that is compatible with a consistent

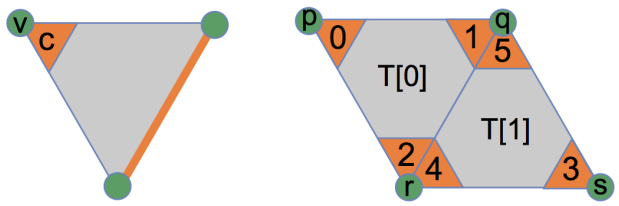


Figure 2: Left: The orange edge $c.e$ is the opposite edge of corner c. Right: Corners 0 and 3 are opposite corners: $O[0] = 3$ and $O[3] = 0$, i.e. opposite edges $0.e$ and $3.e$ are the same.

G	x	y	z
p	x_p	y_p	z_p
q	x_q	y_q	z_q
r	x_r	y_r	z_r
s	x_s	y_s	z_s

C	V[c]	O[c]	
0	p	3	T[0]
1	q	1	
2	r	2	
3	s	0	T[1]
4	r	4	
5	q	5	

Figure 3: Corner Table for Fig. 2, right. Geometry table (left), Vertex Opposite table (right). Since corners 0 and 3 are opposites of each other, $O[0]$=3 and $O[3] = 0$. A corner c with no opposite corner (red) may be easily identified because we set $O[c] = c$.

orientation of the mesh. Hence, each entry to the $V[c]$ table represents a corner c associating a triangle t with a bounding vertex. The O Table stores the integer reference of the opposite corner, where an opposite corner is a corner in an adjacent triangle that shares the same opposite edge. These ideas have been illustrated in Fig. 2 and Fig. 3.

The *Extended Corner Table (ECT)* extends the Corner Table by storing a reference for the result of the $v.c$ corner operator in a table. The $v.c$ operator provides an arbitrary corner incident on a vertex, enabling constant-time access from a vertex to a corner (equivalently the triangle). The total storage cost for ECT is 6.5 *rpt* as storing $v.c$ costs 0.5 *rpt* as there are roughly half the number of vertices as number of triangles.

As the Extended Corner Table provides a representation and the support of corner operators, we use the Extended Corner Table for the construction of the data structures proposed in this dissertation.

2.3.1 O Table construction

The O Table may be computed in $O(n)$ space and time from the V Table. We first describe a $O(n \log n)$ solution. For each corner c, we make an entry (v_1, v_2, c), where v_1=min($c.n.v$,$c.p.v$) and v_2=max($c.n.v$,$c.p.v$). We sort them lexicographically by (v_1, v_2). Pairs of consecutive entries, (v_1, v_2, c) and (v_1, v_2, d) identify opposite corners: $O[c] = d$ and $O[d] = c$. Alternatively, the O Table may be computed in expected $O(n)$ space and time by using hashing on (v_1, v_2). Two corners c and d can be identified as opposite corners, in expected constant time, if they share the same hash value defined by edge (v_1, v_2). An alternative $O(n)$ space and time technique was proposed by Ueng and Sikorski [44]. The entries are placed in m buckets, where the bucket IDs are identified by v_1. Then the entries in each bucket are placed in a second set of m buckets, where the bucket IDs are now identified by v_2. The corners of two entries that are placed in the same second bucket are identified as opposite corners.

CHAPTER III

PRIOR ART

In this chapter, we review relevant prior art, organized into four categories: Compression techniques, Theoretical limit, General representation and Specialized Triangle Mesh representations. In the categories General representation and Specialized Triangle Mesh representations, we present relevant prior art data structures and report their storage cost assuming RAT compatibility.

3.1 Compression techniques

Various connectivity compression techniques have been proposed to reduce storage. We first discuss Rossignac's Edgebreaker [38]. Edgebreaker encodes the connectivity of the mesh with a string of five symbols {'C', 'L', 'R', 'E', 'S'}. The symbols are generated when visiting triangles in a depth first traversal of the Triangle Spanning Tree. Each symbol corresponds to a unique triangle, and is generated during compression when visiting the triangle via one of its edges. The edge is called a *gate*. For the triangle being visited, the vertex not bounding the gate is called the tip vertex. The vertices are ordered in the way they are visited by the 'C' triangles. We briefly describe the five symbols:

- 'C': The tip vertex was not previously visited, therefore the tip vertex is added to the end of a list of vertices. The tip vertex acquires an ID which is its position in the list of vertices. The traversal continues onto the triangle on the right side.
- 'L': The triangle on the left has been visited, therefore the ID of the tip vertex is known. The traversal continues onto the triangle on the right side.

- 'R': The triangle on the right has been visited, therefore the ID of the tip vertex is known. The traversal continues onto the triangle on the left side.

- 'S': The tip vertex has been previously visited. In this case, the unvisited edge-connected components of the triangle mesh gets divided into multiple components. The triangle on the left side is pushed onto a stack so that the traversal can return to it after visiting the unvisited component on the right side. The traversal continues onto the triangle on the right side. In the 'S' case, the ID of the tip vertex, or more specifically, an offset to the tip vertex ID is needed. In a related work, in Gumhold's Cut Border Machine [20] which is similar to Edgebreaker, they explicitly store the offset. As each offset requires up to $\log(m)$ bits, storing it explicity can be expensive. Edgebreaker eliminates the need to store the offset and the authors prove that the offset can be computed from the Edgebreaker string itself.

- 'E': Both the triangles on the left and right have been visited, therefore the ID of the tip vertex is known. In this case, the traversal continues to the triangle on the top of the stack.

Rossignac [34] proved a storage upper bound of 2 *bpt*. This bound was improved by King and Rossignac [28] to 1.84 *bpt* by observing that the consecutive symbols, 'CL' and 'CE', are impossible. Gumhold [19] further improved the bound to 1.78 *bpt* by observing that some longer sequences of symbols are impossible, e.g. the sequence 'CRE'. In practice, using Huffman encoding predictions, Edgebreaker achieves storage results of less than 1 *bpt* [33].

Touma and Gotsman [42] proposed another connectivity compression technique. Their method is a valence-based method. It exploits the regularity of vertices in triangle meshes. In their work, the triangles are visited in the same order as in Edgebreaker. To explain their work, we utilize the Edgebreaker symbols. When a 'C'

symbol is encountered, they encode the valence of the tip vertex. When an 'S' symbol is encountered they encode an offset, as in the Cut Border Machine [20]. In addition to the offset, they need to encode the number of triangles that are not visited on the right hole of the tip vertex. One thing to note is that as the valence v for the tip vertex is provided, when $v-1$ triangles incident on the vertex have been visited, the remaining one triangle need not be encoded and can be inferred for free. Therefore, in such cases, there is no need to encode the 'L', 'R' or 'E' symbols. In their work, they report an average storage result of 0.75 *bpt*.

These compression schemes provide efficient storage results but incur a significant performance penalty when accessing vertices or triangles in a non-sequential manner. As the meshes are encoded sequentially, if we access a triangle t that is at the end of the sequence, these methods sequentially decompress the entire mesh before retrieving information about triangle t, resulting in slow performance. Therefore, to provide random access in constant time, the entire mesh has to be decompressed and kept in memory.

To address the issue of having to decompress the entire mesh when accessing any element, Yoon and Lindstrom [46] propose a partial decompression solution. In their work, the mesh is still encoded sequentially, but they group sequences of a fixed number (a few thousand) of triangles into clusters. Each cluster is compressed independent of each other. When a random access call to a specific triangle is made, the cluster containing the triangle is identified, then the triangles in the cluster are decompressed. The decompressed triangles of the cluster along with their adjacency information are cached in-core. In general multiple pages are cached, and some pages are paged out when there is not enough room to hold all pages. In their work, they report an average storage result of 4.03 *bpt*, but as the partial decompression scheme is quite involved, the method is relatively slow.

Yoon and Lindstrom's work share similarities with Zipper. Similar to their clusters, in Zipper we also group triangles into smaller clusters consisting of 32 triangles. When a specific triangle is accessed, as in Yoon and Lindstrom's work where they identify the cluster containing the triangle, in Zipper the block containing the triangle can be quickly identified and the information for the corresponding triangle can be computed.

3.2 Theoretical limit

Tutte [43] counted the number of all planar triangulations of m vertices. It is given as:

$$N = \frac{2(4m+1)!}{(3m+2)!(m+1)!}$$

The number of bits needed to specify any triangulation is $\log_2(N)$. The amortized cost of this cost per vertex is $\log_2(N)/m$ which is asymptotically about 3.24 bits per vertex, or equivalently 1.62 *bpt*. Note that the meshes our data structures handle need not be planar only, but can have higher genus and a boundary, therefore this result for planar triangulations is only a lower bound.

Castelli Aleardi et al. [15] proposed a theoretical data structure for an optimal representation of the connectivity of planar triangle graphs of size n elements, which has a storage cost of $1.62 + O(\frac{\log\log n}{\log n})$ *bpt*. The second term diminishes when n is arbitrarily large, therefore the storage cost asymptotically matches the entropy of 1.62 *bpt* [43]. They proved that their representation, which decomposes the mesh into small pieces of size $O(\log^2 n)$ and tiny pieces of size $O(\log n)$, could support constant time queries. Also, their representation may be constructed in $O(n)$ time and space. They store the connectivity graph between the tiny pieces and small pieces, and the connectivity of each tiny piece may be represented by an index into a catalog consisting of all possible triangulations of size $O(\log n)$. In practice, as the constants involved in the $O(\frac{\log\log n}{\log n})$ term of their storage cost is fairly large, it is not possible

to achieve 1.62 *bpt*. Although this theoretical formulation was not implemented, the ideas upon which it is based have been explored in Stable Catalogs [11] which store 3.83 *rpt* (see Section 3.4.3).

3.3 General Representations

In this section, we cover general representation schemes for polygonal meshes that can be used to represent triangle meshes. Early representations use much storage, because they cater to more general (polygonal or higher-dimensional) meshes. For each representation scheme, we note their reported storage cost, and also the storage cost required to make it RAT compatible (see Section 1.3). For some of the methods, we describe similarities to our data structures.

3.3.1 Cell-tuple

Brisson's Cell-tuple structure [9], when applied to triangle meshes, associates each triangle t with 6 groupings (k-tuples), each one corresponding to a choice of three entities $\langle v, e, t\rangle$: the triangle t, an edge e of t, and a vertex v of e. There are 6 groupings for each triangle because one has 3 choices for e and then 2 choices for v. With each grouping $g = \langle v, e, t\rangle$, one stores a reference to triangle t, to edge e and to vertex v, plus three references to adjacent groupings known as *swaps*: swap $g.s_0$ returns grouping $\langle v', e, t\rangle$, where v' is the other vertex of e; swap $g.s_1$ returns grouping $\langle v, e', t\rangle$, where e' is the other edge of t that is incident upon v; and swap $g.s_2$ returns grouping $\langle v, e, t'\rangle$, where t' is the other triangle incident upon e. To make Cell-tuple compliant with our definition of RAT, we must store, for each triangle and for each vertex, a reference to one of its groupings. Hence, the total storage cost for connectivity is **31.5** rpt: 6 tuples per triangle that store 5 references each (vertex, triangle, and 3 swaps), plus a tuple reference for each vertex and for each triangle.

3.3.2 Winged Edge

In Cell-tuple, the 4 groupings $g = \langle v, e, t \rangle$ and $g.s_0$, $g.s_2$, and $g.s_2.s_0$ refer to the same edge. The popular Winged Edge representation [6], proposed by Baumgart, combines them into a single edge, with which it associates references to its two bounding vertices, to its two incident triangles, and to the previous and next edge in each triangle. To make Winged Edge compliant with our definition of RAT, we must store a reference to a winged-edge for each vertex and each triangle, so as to support the $v.c$ and $t.c_0$ operators at constant time. Adding these pushes the Winged Edge storage cost to **13.5** *rpt*: 8 references per edge, which is 12 *rpt* (note that the number of edges is 3/2 times the number of triangles), 1 *rpt* for $t.c_0$ and 0.5 *rpt* for $v.c$.

3.3.3 Half Edge

Mantyla's Half-edge representation [31] associates with each half-edge a reference to the next, previous and opposite half-edge, together with a reference to a bounding vertex and incident face for a storage cost of 5 references per half-edge, or 15 *rpt*. Adding support for $t.c_0$ and $v.c$ for their half-edge counterpart yields a total cost of **16.5** *rpt*.

The Surface-Mesh representation [39] extends the Half-edge data structure. In Surface-Mesh, half-edges are reordered such that opposite half-edges are consecutive, therefore half-edges at index $2 * i$ and $2 * i + 1$ are opposite half-edges. As opposite half-edges are consecutive, there is no need to explicitly store the opposite half-edge, which eliminates one reference per half-edge. Additionally, Surface-Mesh does not store a reference to the previous half-edge. Hence, the resulting storage cost is **10.5** *rpt* (as Surface-Mesh eliminates 2 references per half-edge, or 6 *rpt* from the Half-edge representation).

3.3.4 Star-vertices

Kallmann and Thallmann proposed Star-vertices [26]. Star-vertices stores an array of records. Each record, which stores information for a particular vertex, is known as a star-vertex. A star-vertex stores two things: a list of its neighbor vertices sorted in a counterclockwise order, and a count of neighbor vertices. Two vertices are neighbors if they are connected by an edge. We now calculate the storage cost for Star-vertices. They store a pointer to each star-vertex record, which requires 1 reference per vertex, and store a count of the total neighboring vertices which requires an additional 1 reference per vertex. For manifold triangle meshes with low genus, there are on average 6 neighboring vertices per vertex, therefore, storing the neighbors requires an average of 6 references per vertex. Therefore, the total storage cost for Star-vertices is 8 reference per vertex, or equivalently **4** *rpt*.

To make Star-vertices compliant with our definition of RAT, we must add a reference from each triangle to one of its vertices or half-edges (equivalent to $t.c_0$) and a reference from each half-edge to its incident triangle (equivalent to $c.t$), meaning storing one reference per half-edge to an incident triangle (which amounts to 3 *rpt*), and 1 *rpt* for $t.c_0$. The total storage for a RAT compatible Star-vertices representation is **7.5** *rpt*.

Star-vertices shares similarities with SOT and SQuad. In Star-vertices, when inferring adjacent triangle information, the sorted list of ring vertices is traversed. In our SOT and SQuad representations, we use a similar idea: when inferring the vertex IDs, we traverse the triangles incident on the vertex. In Star-vertices, the adjacent vertices are stored, while the adjacent triangles are inferred. In our SOT and SQuad data structures, we do the opposite: the adjacent triangles are stored while the vertex IDs are inferred.

3.4 Specialized Triangle Mesh representations

In this section, we cover representations specialized for triangle meshes. Representations restricted to triangle meshes can exploit the regularity of the connectivity (3 vertices per triangle and 3 neighbors per triangle) and reorder triangles, edges, and/or vertices. As in the general representation section, for each representation scheme, we report the storage cost, and also the storage cost required to make it RAT compatible. Also, for some of the methods, we describe similarities to our data structures.

3.4.1 Corner Table

The Corner Table (CT) promoted by Rossignac et al. [35] provides a simple and efficient representation of triangle meshes, storing 6 integer references per triangle (3 vertex references in the Vertex Table and 3 references to opposite corners in the Opposite Table). As discussed in Section 2.3, the Extended Corner Table (ECT) stores a reference per $v.c$ corner operator resulting in **6.5** *rpt*. The Corner Table is described in detail in this dissertation in Section 2.3.

Our work is derived from the principles defined in the Corner Table. In our work, we use corners and corner operators defined in the Corner Table. Our work can be viewed as providing an implementation for the Corner Table API, but with reduced storage cost.

3.4.2 Directed Edge

Campagna, Kobbelt and Seidel proposed the Directed Edge representation [10], which is a specialization of the Half-edge data structure. It is equivalent to the Corner Table [35], when considering a bijection between half-edges and corners. Like the ECT, the Directed Edge representation uses **6.5** rpt when augmented with the half-edge equivalent $v.c$ references.

3.4.3 Stable Catalogs

Castelli Aleardi, Devillers and Mebarki proposed Stable Catalogs [11]. In Stable Catalogs, a catalog contains entries describing unique connectivity configurations consisting of small groups of up to 7 triangles. Triangles are grouped into patches and, instead of storing the internal connectivity of each patch, a reference to a catalog entry that defines the connectivity of the patch is stored. To address adjacency across patches, information for the boundary of the patch is stored, where for each boundary element (edge or vertex) of a patch, a reference to the boundary element on the neighboring patch is stored. Patches consisting of 1 through 7 triangles are explored in the paper. In Stable catalogs, several versions, each consisting of different catalogs, is explored. These result in different storage results. The most compact version of Stable Catalogs uses **3.83** *rpt*.

Our data structures share similarities with Stable Catalogs. In Stable Catalogs, triangles are grouped into patches where internal connectivity in the patch need not be stored. This grouping saves storage. We use the same principle in our work. In SQuad, LR, and Zipper, we create patches consisting of two adjacent triangles, which we call quads. The adjacency information for triangles in the same quad need not be stored.

3.4.4 Tripod

Snoeyink and Speckmann propose Tripod [40] which computes Schnyder woods to orient the edges and assigns each edge into one of 3 sets, where each set defines a rooted oriented vertex spanning tree. Each vertex has 3 outgoing edges, each of which belongs to a different oriented vertex spanning tree. For each such outgoing edge from a given vertex, Tripod stores the references to the previous and next edges when the edges incident on the vertex are sorted around the vertex. As Tripod stores 2 references per outgoing edge, the storage cost is 6 *rpv* or equivalently **3** *rpt*.

To make Tripod RAT compatible, we add a reference for $t.c_0$ and a triangle ID with each half-edge (equivalent to $c.t$) bringing the total cost to **7** *rpt*.

Tripod shares similarities with our work: LR and Zipper. In Tripod, the adjacent triangle information is not always stored explicitly. Instead, a small set of local triangle configurations are checked to infer adjacent triangle information. These checks translate to a small set of if-else conditions. We use a similar idea in LR and Zipper. In most cases, we do not explicitly store the adjacent triangle information, and like in Tripod, we check a small set of local triangle configurations to infer adjacent triangle information.

3.4.5 Sorted Tripod

Castelli Aleardi and Devillers propose Sorted Tripod [12] which extends Tripod by sorting the vertices. The vertices are sorted according to the Depth First Unary Degree Sequence (DFUDS). To generate DFUDS, first take one of the oriented vertex spanning trees and number the root as 0, then number the root's children, in counterclockwise order e.g. if the root has k children, the children's IDs are numbered 1 through k. After the root and its children are numbered, then the children for the vertex with the next smallest ID are numbered. The children are numbered consecutively in a counterclockwise order. Then the children of the vertex with the next smallest ID are processed, and so the process continues. The sorting of vertices in DFUDS enables vertices in Sorted Tripod to infer information based on the adjacent vertex IDs and local triangle configurations, therefore Sorted Tripod stores fewer references than Tripod while inferring the missing ones. As in Tripod, for one of the outgoing edges, 2 references to the previous and next edges are stored, but for the remaining two outgoing edges, Sorted Tripod stores 1 reference each. Therefore, the storage cost is 4 *rpv* or **2** *rpt*.

To make Sorted Tripod RAT compatible, as in Tripod, we add a reference for $t.c_0$

and a triangle ID with each half-edge bringing the total cost to **6** *rpt*.

As in Tripod, Sorted Tripod shares similarities with our data structures. The first similarity is that in LR and Zipper, as in Sorted Tripod, the adjacent triangle information need not be stored, but can be inferred. The second similarity is the use of the sorting principle. In Sorted Tripod, the vertices are sorted, and the sorting enables Sorted Tripod to infer some of the information to be inferred instead of explicitly storing it. We utilize sorting in all our work. In SOT and SQuad, we sort the triangles, which enables the vertex IDs to be inferred. In LR and Zipper, we sort the triangles *and* vertices, which enables some of the vertex IDs to be inferred without having to store them.

CHAPTER IV

SOT

4.1 Introduction

In this chapter, we discuss SOT. We start with a brief overview. The Sorted Opposite Table (SOT) representation reduces the storage requirement for the connectivity graph to 3 *rpt* (integer references per triangle). We achieve this by matching each vertex with a unique triangle, and by reordering the triangles (but not the vertices) so that part of the connectivity graph can be inferred at run time. We present a linear-time construction algorithm. SOT has been extended to tetrahedral meshes [24].

Sorted Opposite Table (SOT) matches each vertex with a different triangle and reorders triangles so that triangle i of the first m triangles corresponds to vertex i. Hence, there is no need to store the incidence table, which may be recovered by swinging around each vertex until a triangle is reached with a sufficiently small identifier. The $v.c$ operator is also available implicitly. Hence SOT uses 3 *rpt*.

In the following subsections, we discuss three improvements that SOT offers over the Corner Table.

1. Vertex to corner access: To provide a constant time access, we reorder the triangles and the corners within triangles so that for the first m triangles, the first corner of the i^{th} triangle corresponds to the i^{th} vertex (see Fig. 7). We call this data structure the Sorted Vertex Opposite Table (SVOT). We discuss this in Section 4.2.

2. 3 references per triangle: To further reduce storage, we discard the V Table in SVOT. The resulting Sorted Opposite Table (SOT) stores only 3 references per triangle. We discuss this in Section 4.3.

3. Constant time operators: We use the same operators as the original corner table, except for the v operator (vertex operator). We discuss this in Section 4.3.1.

4.2 Sorted Vertex Opposite Table(SVOT)

4.2.1 Motivation

The SVOT solution described below provides the same information as the Extended Corner Table (ECT). In ECT, the $v.c$ operator is stored as a lookup table. The SVOT solution avoids storing this lookup table yet provides the $v.c$ operator in constant time.

4.2.2 Proposed SVOT solution

To provide constant time access to a corner $v.c$ for each vertex v, and this without additional storage, we reorder the triangles and their corners in the CT so the corner $v.c$ incident upon vertex v may be simply computed as $3v$. This mapping works for most meshes. However, for meshes with *narrow* components, we may need a slightly more complex special mapping, as shown in Fig. 4 which we discuss in Section 4.4. A mesh is *narrow* if each triangle in the component has at least one *border vertex*. A *border vertex* is a vertex that lies on the boundary of a mesh.

4.2.3 SVOT construction algorithm

We explain here how to compute SVOT from the CT in linear time. The process involves 2 steps: (1) Vertex to Triangle Matching: Establish a matching, $M[t] = v$, between each vertex v and an incident triangle t so that no two triangles match to the same vertex. (2) V Table Sorting: Reorder the CT based on the matching.

4.2.3.1 Vertex to Triangle Mapping

The matching phase involves three steps: (i) Initialization, (ii) Traversal, and (iii) Termination.

To represent the matching M computed in the matching phase, we use a temporary table M which stores the vertex number $M[t]$ associated with triangle t. We use three

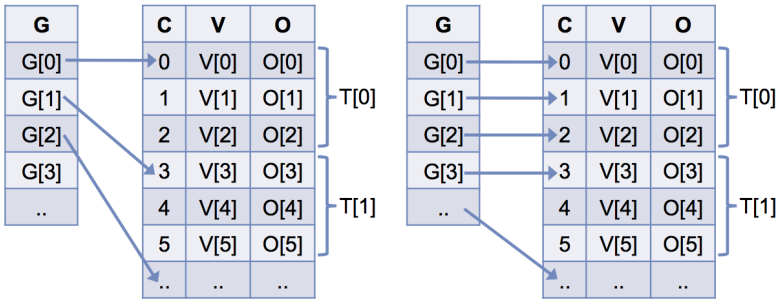


Figure 4: Left: General SVOT mapping. Right: Special mapping for narrow components. After the sorting, the V Table may be discarded to obtain the SOT.

arrays of auxiliary tables: $visitedV[v]$ keeps track of visited vertices v, $visitedT[t]$ keeps track of visited triangles t, $whichCorner[t]$ stores the corner c in triangle t such that $M[c.t\ ]$=$c.v$.

We traverse the Triangle Spanning Tree (TST), which is the set of triangles visited in depth first order from an initial triangle. As we enter a new triangle t through an edge e, we associate t with its *tip vertex* (the vertex in triangle t not bounding edge e), unless that vertex has already been visited and hence associated with another triangle. This idea is similar to the association of the tip vertex of each type-C triangle in the Edgebreaker compression scheme [35] for triangle meshes. Unfortunately, unless we take special precautions, this simple idea may not always work, because the traversal may associate each triangle incident upon a vertex v with a vertex other than v, leaving some vertices unmatched to any triangle. To eliminate the possibility of unmatched vertices, we perform a special initialization step, which guarantees that this approach produces a correct matching. A correct matching is one where all vertices are matched to triangles such that each vertex is associated with a unique triangle.

(i) Initialization: During the initialization step, we pick a seed triangle S so that none of its vertices bound a border edge. Let c be the first corner of S. We set all entries in the array M to be -1 denoting unmatched triangles. We set $M[S]$=$c.v$ and

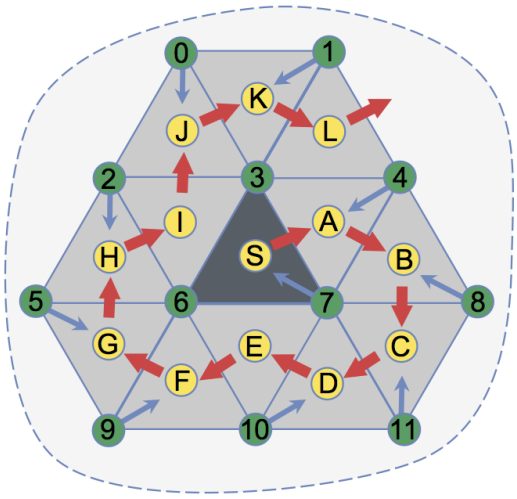


Figure 5: Depth first traversal of triangle mesh starting from seed triangle S. Red arrows represent traversal order, blue arrows represent matching of vertices to triangles. Notice vertices 6 and 3 are marked as visited in the initialization step. Also because a non-border seed triangle S is chosen (i.e. all its vertices are interior), triangles E and I are unmatched.

mark (as visited) all vertices of S.

In Fig. 5, the seed triangle S is the dark gray triangle with label S. S is bounded by vertices 7, 6 and 3 and Vertex 7 is $c.v$. Vertices 7, 6, 3 are marked as visited and $M[S] = 7$.

Note that this approach assumes that a suitable seed exists. Finding S, when it exists is trivial. The approach proposed above works for edge-connected components of meshes that are not narrow. It picks as seed a triangle with no border vertex. For narrow components of meshes, we use a slightly modified solution described in Section 4.4. For multiple edge-connected components, we need multiple seed triangles. After selecting a seed triangle S, we mark S and start a depth first traversal of the triangle spanning tree with S as root and $c.l.t$ as the first child, where c is the first corner of S.

In the initialization step, in Fig. 5, $visitedV[7]$, $visitedV[6]$, $visitedV[3]$ and $visitedT[S]$ are set to $TRUE$. For example, if the corner of S bounded by vertex 7

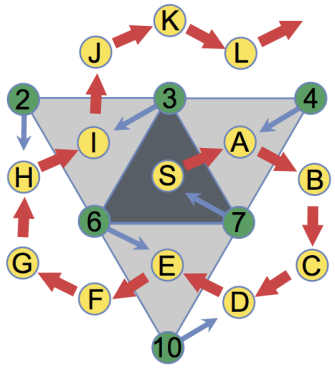


Figure 6: A portion of Fig. 5. In Fig. 5, vertices 3 and 6 are unmatched, and triangles E and I are unmatched too. We introduce matchings $(6, E)$ and $(3, I)$. Due to the nature of depth first traversal, and a internal triangle S, this matching is always possible.

was 30 then $whichCorner[S]$ would be assigned 30.

(ii) Traversal: During the traversal step, in our depth first order traversal, we reach a new unvisited triangle t by arriving from a parent triangle through the opposite edge $b.e$ of a corner b (of triangle t). We mark t as visited. If the vertex $b.v$ has not yet been visited, we mark it as visited, set $M[t]$=$b.v$, and store in $whichCorner[b.t]$ corner b.

For example, in Fig. 5, the first triangle visited after triangle S is triangle A. We arrive at triangle A through edge (3,7). Let b be the corner bounded by vertex 4. Since vertex 4 was previously not visited, therefore, $M[A] = 4$ and $whichCorner[A] = b$.

(iii) Termination: We match the two triangles E and I (see Fig. 6) adjacent to seed S (which are $c.n.l.t$ and $c.p.l.t$ where c is the first corner of S) with two vertices 6 and 3 of S (which are $c.v$ such that c is not the first corner of S), as shown in Fig. 6. If the seed triangle is an internal triangle, then S has three adjacent triangles. The other two triangles E and I adjacent to S have been reached while coming from triangles other than S and hence will not be associated with a vertex (since their tip vertices 6 and 3 were marked as visited during initialization and is no longer available to be associated with them).

When we perform a depth-first traversal starting from S, we visit all triangles in

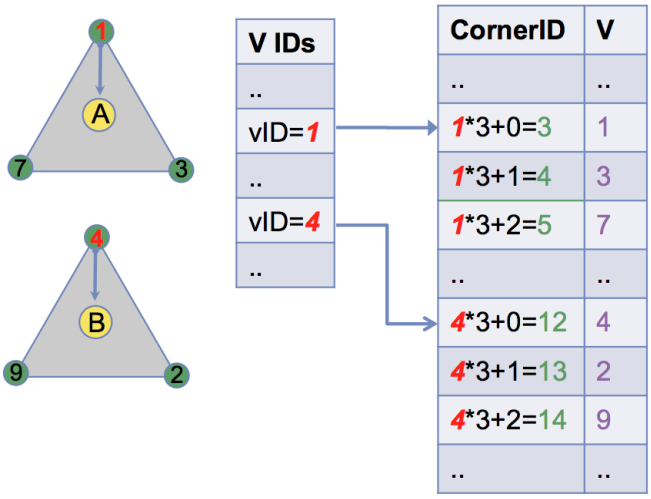


Figure 7: Vertices 1 and 4 are matched to triangles A and B. Triangle A consists of vertices (1,3,7) and triangle B of (4,2,9). After sorting in SVOT, triangle A is defined by corners (3,4,5) and triangle B by corners (12,13,14) as vertex 1 maps to triangle at 1^{st} location and vertex 4 maps to triangle at 4^{th} location. Note that vertex 1 is mapped to the first corner of triangle A, and vertex 4 is mapped to the first corner of triangle B.

the connected component S is part of and also all the vertices. Other than the seed triangle S and its incident vertices, which we addressed above, each time we visit a vertex, we match the vertex to the triangle that visits the vertex. Therefore each vertex is matched to a unique triangle.

4.2.3.2 Sorting the V Table

We write the triangles into a new copy of the V Table ensuring that triangle t from the old V Table is listed as triangle number v in the new V Table, where $v = M[t]$. We perform a cyclic permutation of the corners of each triangle so that the corner stored in $whichCorner[t]$ is listed as the first corner of t. Note that about half the number of triangles are not matched to any vertex. In the new copy of the V Table, these unmatched triangles are placed in an arbitrary order after the first m matched triangles. Since we have changed the ordering of the entries of the V Table, the O Table references are no longer correct, so we re-compute the O Table (as described in

Section 2.3.1). Fig. 7 illustrates the V Table of the resulting SVOT.

The sorting discussed has linear time and space complexity in the number of triangles. The sorting needs to be performed only once, since its result (i.e. the sorted V Table) may be archived for future uses. This sorting has linear time complexity i.e. $O(m)$ instead of the traditional $O(m \log m)$ that is associated with sorting, as this sorting is a permutation where each element knows the index in the array it has to be in in the sorted table.

4.2.4 Traversing the star

Here, we describe the algorithm to traverse the star of a vertex. This algorithm will be used when computing the $c.v$ operator in SOT in Section 4.3.1.

Given an integer reference v to a vertex, the SVOT gives us direct access to the corresponding corner $v.c$, using $v.c = 3v$.

The i^{th} vertex is mapped to the 3*i^{th} corner in the SVOT. Notice, in Fig. 7, in the SVOT, vertex 1 is located in the corner location 3*1=3, and vertex 4 is located in the corner location 3*4=12. To visit the star of the i^{th} vertex, we simply call star(3*i) and we can traverse all incident triangles on a vertex by iteratively using the swing and unswing corner operators. The $star(c)$ function for corner c, which assumes that vertex $c.v$ is interior, is listed below.

```
void star(int c) {
  int sc = c;
  do {
    c = c.s;
  } while(c!=sc);
}
```

If $c.v$ is a border vertex, we need to keep track of boundary corners (opposite edges of corner is a boundary edge) and use both the swing and unswing corner operators. The function is listed below.

```
void star(int c) {
  int sc = c;
  do {
    c = c.s;
  } while(c!=-1);

  c = sc;
  do {
    c = c.u;
  } while(c!=-1);
}
```

The corner operators for the Corner Table work without modification on the SVOT.

4.3 Sorted Opposite Table (SOT)

The Corner Table and our SVOT variation each store 6 references per triangle (3 to vertices, 3 to opposite corners). We discuss here an approach to reduce this storage to 3 references per triangle and store the result in the Sorted Opposite Table (SOT) data structure. The resulting SOT contains no references to vertices. How then is it possible to find vertex references $c.v$ of a corner c? The solution comes from a combination of three ideas.

1. Because the O Table is sorted in the SVOT, to each vertex v corresponds a matching triangle t of which the first corner is incident on v. Note that such triangles are easily recognized because their index t is less than the number m of vertices. A slightly modified mapping relation is used for narrow components (See Section 4.4 for narrow components).

2. By construction of the SVOT, in the star of every vertex v, there is a matching triangle t.

3. Starting from any corner c, we can use the star function provided in Section 4.2.4 to visit the star of its vertex $c.v$, even though we do not yet know the index of v.

The idea is to traverse the star (see Section 4.2.4) of $b.v$ to determine the corners b_i incident on $b.v$. We must do that of course without knowing $b.v$, since $b.v$ is the desired result. Traversing the star is possible by using the swing and unswing corner operators as these corner operators require only the O Table (i.e. connectivity information, not the V Table) of the SOT. The traversal stops when we find a matching triangle $t < m$.

Finding the vertex reference can require that we visit at most d triangles, where d is the valence of the vertex v. Therefore, finding the vertex reference has $O(d)$ time complexity. The valence of a vertex on a triangle mesh with low genus and no boundary is approximately 6. Therefore, on average, finding the vertex reference has constant time complexity.

4.3.1 Corner operators on SOT

The implementation of all corner operators from the Corner Table remain the same in SOT, except for the v operator. The v operator in SOT traverses the star of vertex $c.v$ without knowing $c.v$ to determine the vertex reference. The code for the $c.v$ function is listed below:

```
int v(int c) {
  int sc = c;  //save starting corner
  int cc = c;
  do {
    if(c.t<m && c % 3==0) { //first corner of triangle, and
                            //one of first m matched triangles
      return c.t;
    }
    cc = c;
    c = c.s;
```

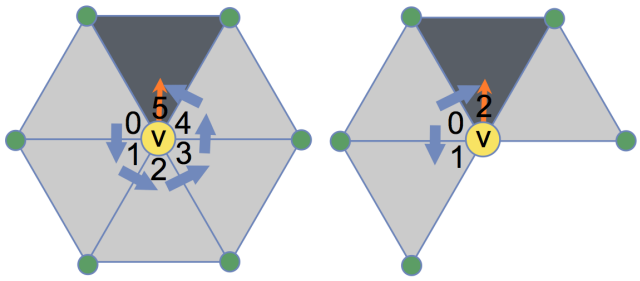


Figure 8: To determine the vertex ID of the vertex v, we traverse the incident triangles by using the swing corner operator, s or u. In the SOT, one of the incident triangles has the i^{th} vertex matched to the i^{th} triangles first corner (yellow vertex and orange arrow).

```
  } while(c!=sc || c!=cc); //back to starting corner,
                           //or boundary corner


  c = sc;
  do {
    if(c.t<m && c % 3==0) { //first corner of triangle, and
                           //one of first m matched triangles
      return c.t;
    }
    c = c.u;
  } while(true); //back to starting corner,
                           //or boundary corner
}
```

Starting from a corner c, the v operator pivots clockwise and then, if necessary, counterclockwise around $c.v$ using the $c.s$ and $c.u$ corner operators. It stops and returns $c.t$ if it finds a corner c such that $c.t < m$.

The small overhead cost of the new v operator function, detailed above is often justified by the reduction of storage, and hence of page faults when ECT does not fit in memory.

4.4 Special cases of narrow components

Here, we discuss the special case for narrow components. A mesh is narrow if each triangle in the component has at least one border vertex. For each narrow component, we choose any triangle as the seed S. We force S to be the first triangle in the V Table. Let the three vertex references for S be v_0, v_1 and v_2. We swap the vertices in the geometry table where we swap vertex v_0 with the 0^{th} entry in the geometry table, vertex v_1 with the 1^{st} entry in the geometry table and likewise for v_2. We then reorder the rest of the triangles as described in the construction section for the general case. Now, the vertex-to-corner mapping is as follows: for i<3, the i^{th} vertex maps to the i^{th} corner. For $i \geq 3$, the i^{th} vertex maps to the $((i-2)*3)^{th}$ corner. Correspondingly, for meshes with q narrow edge-connected components, we can place each seed triangle for each component as the first q triangles in the V Table. This special mapping has been shown in Fig. 4, right.

CHAPTER V

SQUAD

5.1 Introduction

In this chapter, we discuss SQuad. We start with a brief overview. The Sorted Quad (SQuad) representation reduces the storage requirements for the connectivity graph to about 2 *rpt* (integer references per triangle). We achieve this by matching pairs of adjacent triangles with one of their shared vertices, and by reordering the triangles (but not the vertices) so that the connectivity graph can be inferred at run time. We present a linear-time construction algorithm and describe an optimized implementation of the operators that may be used to traverse the mesh efficiently using only its SQuad representation.

The ability to preserve the vertex order is a significant strength of our representation. For instance, it is important for stream processing, for enforcing data locality, and for applications that for other reasons impose a vertex order.

5.2 Quad meshes

One may easily extend the "triangle" Corner Table to a "quad" Corner Table for representing irregular quad meshes. This can be done by storing information for the four corners of a quad. To distinguish quad corners from triangle corners, we capitalize the names of quad corners and their operators. For efficiency and clarity, instead of storing the opposite corner in table O, we use an S table that stores the swing operator $C.S$, as described in Section 2.2. The following primary quad corner operators (Fig. 10) may be used to traverse the quad mesh:

- $C.V$ returns the vertex of corner C.

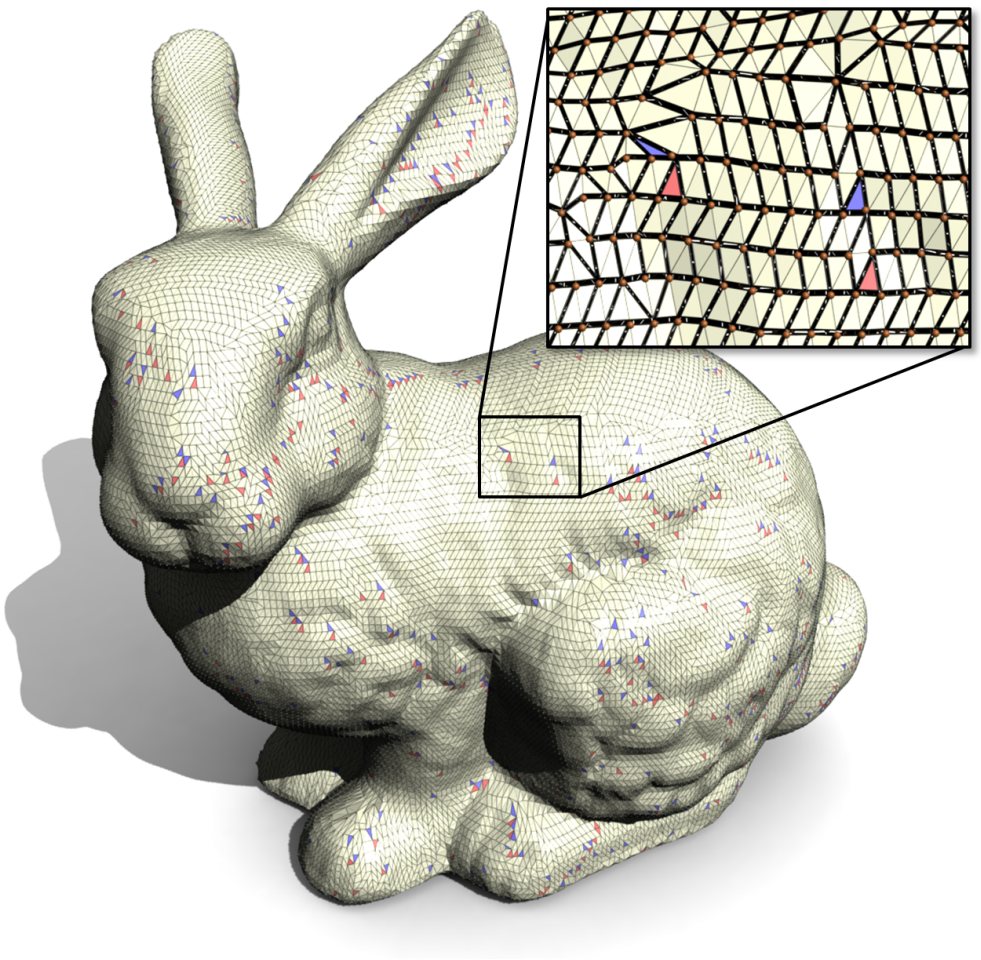


Figure 9: SQuad pairs most triangles (here 97.3%) into quads and matches each vertex with a different quad or single triangle. By sorting the quads and single triangles to match the order of the vertices, one may represent the connectivity of the mesh by storing only 2.05 references per triangle. The rare single triangles are colored blue (unpaired) or red (unmatched).

- $C.Q$ returns the quad of corner C.
- $C.N$ returns the next corner in $C.Q$.
- $C.S$ returns the "swing" corner around $C.V$.
- $V.C$ returns one corner so that $V.C.V = V$.
- $Q.C$ returns one corner so that $Q.C.Q = Q$.

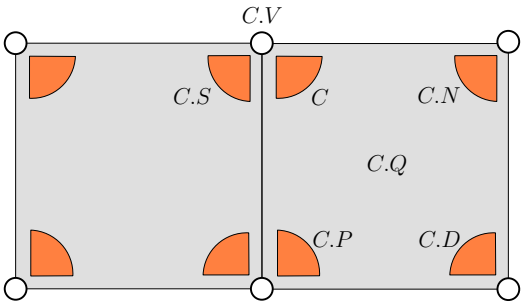


Figure 10: From a corner C, we can access its vertex $C.V$ and quad $C.Q$, the next corner $C.N$ in $C.Q$, and the swing corner $C.S$. For convenience, we also define $C.D$ as $C.N.N$ and $C.P$ as $C.D.N$.

Note that in a mesh of n_Q quads, the V and S tables each have $4n_Q$ entries. Hence, this representation uses 8 *rpq* (references per quad).

5.2.1 Representing triangles with quads

The Euler-Poincare characteristic leads to the identity $n = 2m - 4 + 4g$ (see Section 1.4.1) for a manifold triangle mesh with genus g, n triangles and m vertices, and no boundary implies that n is even. If we arrange triangles into pairs so that each pair shares a common edge and hence forms a quad, we may use the above quad data structure to represent the connectivity of the triangle mesh. This approach translates into 4 *rpt*, since there are two triangles per quad and storing a quad requires 8 *rpq*. We need to add one reference per vertex (i.e. 0.5 *rpt*) for storing $V.C$.

Hence, if we paired all the triangles of a mesh, we could encode its connectivity using 4.5 rpt [13]. The pairing is always possible, since the dual of the mesh is a bridgeless trivalent graph [32], and can be computed in $O(n \log^4 n)$ time [7]. Tarini et al. [41] present a method for converting a triangle mesh into a pure quad mesh. Unfortunately, this approach cannot be used directly for our purpose, unless we find a way to ensure that the pairing makes it possible to match each vertex with a different quad. So we will opt for incomplete pairing.

5.3 SQuad overview

The *SQuad* representation proposed here combines two ideas:

1. the use of a quad mesh to represent the connectivity of a triangle mesh, and

2. the sorting used in SOT discussed in Section 4.2.3.

SQuad requires only a table of swings (the S table) for the quad corners of the mesh to implement a complete set of adjacency queries for both the quad mesh and the triangle mesh. SQuad uses only 4 *rpq* (references per quad), and if most triangles are paired, the storage approaches 2 *rpt*. More precisely, the storage required is $2 + 2f$ *rpt*, where f is the fraction of unpaired or unmatched (*single*) triangles.

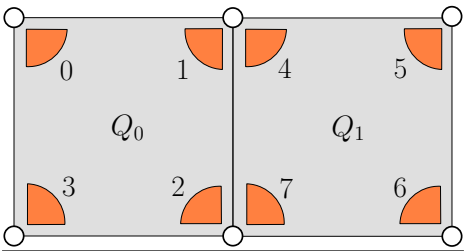
Our construction of SQuad which attempts to match each vertex with an adjacent pair of incident triangles involves the following sequence of steps.

1. In a depth-first traversal (discussed in Section 4.2.3) of the triangle adjacency graph [35], we match each vertex with the triangle that visits it first and attempt to pair that triangle with one of its not yet paired neighbors.

2. Then, we store each pair of triangles as a quad and, for regularity of representation, we also disguise the few unpaired triangles as quads by storing a sentinel value for the fourth unused corner.

3. Finally, we reorder these quads (similarly to SOT) so that, after reordering, the i^{th} quad is the one matched with the i^{th} vertex.

We have tested SQuad on a benchmark of meshes of different complexities, ranging from a few thousand to about 55 million triangles, and report statistics on storage size and on construction and access time. In particular, we found that SQuad storage averages 2.072 rpt on these meshes.



	Q_0				Q_1			
C	0	1	2	3	4	5	6	7
$S[C]$	0	4	7	3	1	5	6	2

Figure 11: S table for a mesh of two quads.

5.4 Representation and operators

Here, we provide the details of the SQuad data structure and of an efficient implementation of its operators. We begin by describing a complete quad mesh representation, and later elaborate on the additional operators needed to support triangle meshes.

5.4.1 Quad mesh representation

SQuad stores only the S table of swings of all the quad corners in the mesh. That is, the swing $C.S$ of a quad corner C is defined as $S[C]$ by indexing the S table (see Fig. 11 for a simple example). The S table is divided into sets of four consecutive entries representing the four corners of a quad. The ID of the i^{th} corner of quad Q is given by $C(Q, i) = 4Q + i$. This consecutive numbering of corners makes the implementation of $C.Q$, $C.N$, $V.C$, and $Q.C$ particularly straightforward, as these operators can be computed using multiplication, division, and modulo by 4.

5.4.2 V operator

What remains is the computation of $C.V$. As discussed above, each vertex V is associated with exactly one quad Q and one of its corners C. In particular, we associate V with the zeroth quad corner $C(Q, 0)$. Thus, we may determine if a corner C is matched with a vertex by examining its two least significant bits. If $C \bmod 4 = 0$, then C is matched with $V = C/4$. Otherwise, we swing around V using $C.S$ until we

find the corner matched with V.

5.4.3 Corner operators

The set of operators defined on a quad mesh can thus be implemented as follows:

$$V.C = 4V$$

$$Q.C = 4Q$$

$$C.V = \begin{cases} C/4 & \text{if } C \bmod 4 = 0 \text{ and } C < 4m \\ C.S.V & \text{otherwise} \end{cases}$$

$$C.Q = C/4$$

$$C.N = 4C.Q + ((C+1) \bmod 4)$$

$$C.S = S[C]$$

The predicate $C < 4m$ is needed when the number of quads exceeds the number of vertices, as in this case some quads cannot be matched with a vertex. If instead the number of vertices exceeds the number of quads, some vertices cannot be matched with a quad, e.g. a tetrahedron has four vertices but only two quads. This case is not handled by our quad mesh representation, but is handled by Meshlets [30], which is an extension of SQuad. Fortunately, such unmatched vertices generally do not occur in triangle meshes, as $n \simeq 2m$.

5.4.4 Corner mapping

To support triangle meshes, we conceptually split each quad along one of its diagonals into two triangles. This splits two corners of each quad in half, while the other two quad corners each map to a single triangle corner (Fig. 12). Aside from this change, we use the same basic data structure, and simply map between triangle and quad corners when necessary. That is, the S table still stores swing pointers between *quad* corners.

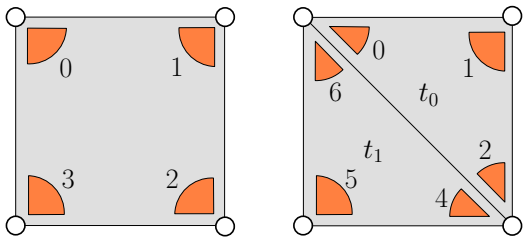


Figure 12: Numbering of corners within a quad (left) and a triangle pair (right).

The mapping $c \mapsto C$ from triangle to quad corners is not a bijection, since pairs of triangle corners may map to the same quad corner. Hence some care is needed in how we perform this mapping. Fig. 12 depicts the mapping that we have chosen between the four quad corners and six triangle corners of a quad. The IDs of the triangle corners of quad Q are given by $c(Q, i) = 8Q + i$, where the corner offsets i are 0, 1, and 2 for the first triangle and 4, 5, and 6 for the second. We do not use offsets 3 or 7; this does not incur storage overhead because we do not store any per-triangle-corner information. While the choice of mapping is not unique, reserving eight offsets per quad enables an efficient implementation based on divisions and modulus operations by 4 and 8, as shown below.

In summary, the mappings (see also Fig. 12) from quad-corners to triangle-corners are: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 4$, $3 \mapsto 5$; and from triangle-corners to quad-corners are: $0 \mapsto 0$, $1 \mapsto 1$, $2 \mapsto 2$, $4 \mapsto 2$, $5 \mapsto 3$, $6 \mapsto 0$. For quad corners associated with two triangle corners, the mapping $C \mapsto c$ is such that the second triangle corner is reached from the first by $c.s$, which as will become apparent ensures that we can traverse all triangle corners around a vertex.

The mappings $c.C$ and $C.c$ may be implemented efficiently without lookup tables. We use the auxiliary functions $C(Q, i) = 4Q + i$, $c(Q, i) = 8Q + i$, $C.Q = C/4$, and $c.Q = c/8$, which allow us to convert from a triangle corner c to a quad corner C:

$$c.C = C(c.Q, (c + ((c/2)\ \&\ 2)) \bmod 4)$$

where '&' indicates the bitwise AND operation.

Before defining the mapping from quad to triangle corners, we note that we may not always be able to pair triangles into quads. Single, unpaired triangles are still treated as quads, and we store for their last corner of a quad in the S table a special sentinel value, referred to as *null* in subsequent sections, to indicate that the second triangle of the quad does not exist. The predicate below determines whether the quad represents one or two triangles:

$$isQuad(Q) = (S[4Q+3] \neq null)$$

We then compute the triangle corner c associated with a quad corner C as follows:

$$C.c = \begin{cases} c(C.Q, (C \bmod 4) + (C \;\&\; 2)) & \text{if } isQuad(C.Q) \\ c(C.Q, (C \bmod 4)) & \text{otherwise} \end{cases}$$

Although $C.c.C = C$, in general $c.C.c \neq c$.

5.4.5 Triangle meshes

We are now ready to define the SQuad operators that enable efficient extraction of the triangle mesh connectivity information. We present their implementation first, and follow with details:

$$v.c = 8v$$

$$t.c = 4t$$

$$c.v = c.C.V$$

$$c.t = c/4$$

$$c.n = \begin{cases} c-2 & \text{if } c \bmod 4 = 2 \\ c+1 & \text{otherwise} \end{cases}$$

$$c.s = \begin{cases} c(c.Q, 6) & \text{if } c \bmod 8 = 0 \text{ and } isQuad(c.Q) \\ c(c.Q, 2) & \text{if } c \bmod 8 = 4 \\ c.C.S.c & \text{otherwise} \end{cases}$$

The standard corner-based operators follow directly:

$$c.p = c.n.n$$

$$c.o = c.p.s.p$$

$$c.l = c.s.p$$

$$c.r = c.n.s.p$$

The majority of our operators have straightforward implementations. We note that $c.v$ is computed efficiently by iteration over quad corners. This allows us to "skip over" triangle corners known not to be matched with v. Due to the consecutive ordering of corners, $c.n$ like $C.N$ is efficiently implemented using modular arithmetic. Finally, $c.s$ computes the adjacent swing corner if it is within the same quad (the first two cases); otherwise it consults the S table. Our operators run in constant time, except $c.v$, which runs in expected constant time but in time linear in the maximum degree in the worst case.

5.5 SQuad construction

Our linear-time construction of the SQuad data structure starts with the V table of a mesh. We compute the O table (see Section 2.3.1) and then match vertices with triangles, pair most triangles into quads, reorder the quads and single triangles, and finally produce the S table. In this section we describe the triangle-vertex matching and triangle-triangle pairing operations, then describe a single-pass algorithm that

combines the two operations. To avoid ambiguity, we use "matching" to refer to triangle-vertex associations and "pairing" to refer to triangle-triangle associations.

5.5.1 Triangle-Vertex matching

Our algorithm computes vertex-triangle matches first, instead of directly matching quads with vertices. This is necessary because some meshes have more vertices than quads. For example, a quad mesh with genus zero has $m = n_Q + 2$ vertices (where $n_Q = n/2$), which shows that two vertices would remain unmatched.

Our matching procedure is described in Section 4.2.3 and illustrated in Fig. 13. We summarize it here for completeness. We start with a seed triangle and match its three vertices according to Fig. 13. We then invade the mesh [35] by walking between edge-adjacent triangles. For the corner c not incident on the previous triangle, if its vertex $c.v$ is unvisited, then we match it with triangle $c.t$. This ensures that each vertex is visited and matched with a different triangle. Since $n \simeq 2m$, only about half of the triangles are matched. We observe that unmatched triangles tend to be uniformly distributed around the matched ones.

5.5.2 Triangle-Triangle pairing

Next, we try to pair each matched triangle with an unmatched one by traversing the triangles in the same order as before. For each matched triangle t, we check first its right, then left neighbors; if we find a neighbor that is neither matched nor paired, we pair it with t (Fig. 13). This simple procedure leaves very few unpaired triangles (see Fig. 9): on average 3.3% of the triangles remained unpaired in our benchmark meshes.

5.5.3 Combined matching and pairing

The matching and pairing process may be implemented in a variety of ways. A particularly elegant implementation uses a modified Edgebreaker [35] traversal of the

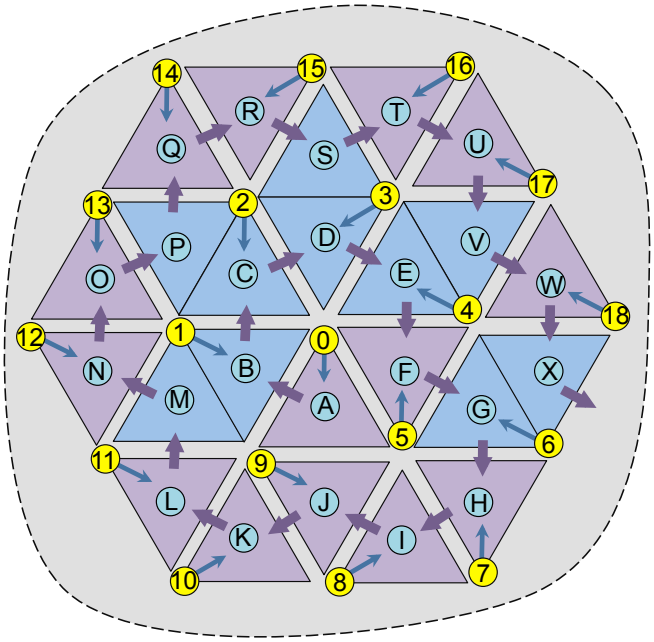


Figure 13: We invade the mesh starting from triangle A. Purple arrows show the traversal order; blue arrows are triangle-vertex matches. Matches $(0, A)$, $(5, F)$, and $(9, J)$ are made at the beginning. Note that triangles M, P, S, V, and X are unmatched and thus paired (blue quads) with B, C, D, E, and G.

mesh, which performs the matching and pairing in a single pass, as shown in Fig. 13. Edgebreaker performs a depth-first traversal of the mesh triangles, and labels them using one of five symbols {'C', 'L', 'E', 'R', 'S'} depending on how the triangles are attached to the already visited triangles (see Section 3.1 and [35] for further details). We include our matching algorithm here, to show its simplicity (see Fig. 14).

Our algorithm takes as input an arbitrary seed corner c, and outputs two tables M and P such that v is matched with triangle $M[v]$ and triangle t is paired with triangle $P[t]$. Thus, the tuple $\langle v, M[v], P[M[v]]\rangle$ forms a vertex-quad match from which the SQuad S table is easily constructed. All entries of M and P are assumed initialized to **null**. Our algorithm makes use of a temporary table T of booleans, which records for each triangle if it has been visited.

From the start corner c, we walk on the mesh while a stack is not empty. Each 'C' triangle $c.t$ involves visiting a new (unmatched) vertex $c.v$, which we match with

```
MATCH-AND-PAIR(in : c, out : M, P)
 1  M[c.v] ← c.t                          match first 3 vertices
 2  M[c.n.v] ← c.n.s.t
 3  M[c.p.v] ← c.p.s.t
 4  P[c.n.s.t] ← c.n.s.t                  mark triangles as paired
 5  P[c.p.s.t] ← c.p.s.t
 6  T[c.t] ← true                         mark c.t as visited
 7  c ← c.l
 8  stack.push(null)                      push sentinel value
 9  while stack ≠ ∅
10      T[c.t] ← true
11      if M[c.v] = null
12          M[c.v] ← c.t                  case C
13          if P[c.r.t] = null and M[c.r.v] ≠ null
14              P[c.t] ← c.r.t            pair c.t and c.r.t
15              P[c.r.t] ← c.t
16          else if P[c.l.t] = null
17              P[c.t] ← c.l.t            pair c.t and c.l.t
18              P[c.l.t] ← c.t
19          c ← c.r
20      else
21          if T[c.l.t] = true
22              if T[c.r.t] = true
23                  c ← stack.pop()       case E
24              else
25                  c ← c.r               case L
26          else
27              if T[c.r.t] = true
28                  c ← c.l               case R
29              else
30                  stack.push(c.l)       case S
31                  c ← c.r
```

Figure 14: Matching & pairing in a single pass.

$c.t$. We then attempt to pair $c.t$ with an unpaired neighbor. Because 'C' triangles generate vertex matches, and because each quad is matched with only one vertex, no two 'C' triangles may be paired; a condition we test for on line 13 in Fig. 14 by making sure that vertex $c.r.v$ has already been matched (otherwise $c.r.t$ is a 'C' triangle). A similar test on line 16 is not needed, since we must enter $c.l.t$ from a triangle other than the current one, and hence $c.l.t$ cannot be a 'C' triangle. This order of trying to pair right neighbors before left ones reduces slightly the ratio of single triangles, because it pairs more of the previously visited triangles. For 'E', 'L', 'R', and 'S' triangles, no matching or pairing occurs, and for those cases we simply follow the Edgebreaker traversal. Similar to the Edgebreaker traversal, for each 'S' triangle we push the $c.l$ corner into the stack, and for each 'E' triangle, we pop the corner from the stack.

5.5.4 Quad reordering

We reorder quads (or single triangles) based on vertex matching, such that the i^{th} quad (or i^{th} single triangle) and its first corner are matched with the i^{th} vertex. Quads of unmatched triangles are placed at the end of the list. The original vertex order is thus left unmodified, while the triangle order is made "compatible" with the vertex order. This freedom in ordering may be exploited in applications that, for instance, require high locality of reference, or for further mesh compression for offline storage.

5.6 Topology extensions

So far, for simplicity and clarity, we have assumed that the mesh is a manifold without boundary. Here, we explain how we have modified our representation to support any orientable, non-manifold mesh, as long as it can be represented as a pseudo-manifold [36] with boundary using a Corner Table.

We store for each boundary vertex v a "bucket" of all corners incident upon v. We then perform the following steps:

1. We partition v's bucket into disjoint subsets of corners. Each set has triangles that form edge-connected components.

2. We sort each set so that all consecutive corners (c_i, c_{i+1}) share an edge, i.e., $c_{i+1} = c_i.s$.

3. For the last corner z of a set, we change $S[z]$, which may have been referencing the first corner or nothing (indicating that z was next to a border edge), to the first corner of the next set (or when z is the last corner in the last set, we set $S[z]$ to reference the first corner in the first set).

As a result, the links stored in S allow us to traverse all of the corners of a vertex v, even when v is non-manifold. However, to implement the true $c.s$, we now need to know (1) whether $S[c]$ refers to the next set (i.e., when $c.t$ and $c.s.t$ are not edge-connected), and (2) whether $c.s$ exists. We use two bits of $S[c]$ to record this information and modify the implementation of $c.s$ accordingly.

CHAPTER VI

LR

6.1 Introduction

In this chapter, we discuss LR. We start with a brief overview. LR (Laced Ring) stores on average 1.08 references per triangle (equivalently 34.6 bits per triangle), and BELR (Bit-Efficient version of LR) stores on average 26.2 bits per triangle. The construction for LR and BELR construction, from an input mesh format that supports constant-time adjacency queries, has linear space and time complexity, and involves ordering most vertices along a nearly-Hamiltonian cycle.

LR supports the full set of standard random-access operators, including all those supported by CT, plus the vertex-to-incident-triangle (star) reference. These operators provide random access from an element (vertex, edge, or triangle) to adjacent elements, and permit visiting the vertices of a triangle and the triangles or edges incident upon a vertex in the cyclic order defined by the orientation of the mesh. We provide the details of a practical and efficient implementation of these operators, which each have constant-time complexity.

This significant progress over prior art builds on the following novel contributions.

Ring-based ordering: We build a nearly-Hamiltonian cycle of edges that we call the **ring**. It divides the mesh in two parts (Fig. 15) that form triangle strip corridors with bifurcations (Fig. 15). We classify triangles by the number of edges they have on the ring (**bifurcation** T_0, **normal** T_1, **dead-end** T_2). We store the ring vertices and the T_1 and T_2 triangles in the order in which they are visited by the ring. The isolated vertices not part of the ring are stored last. The T_0 triangles are stored using the standard CT data structure.

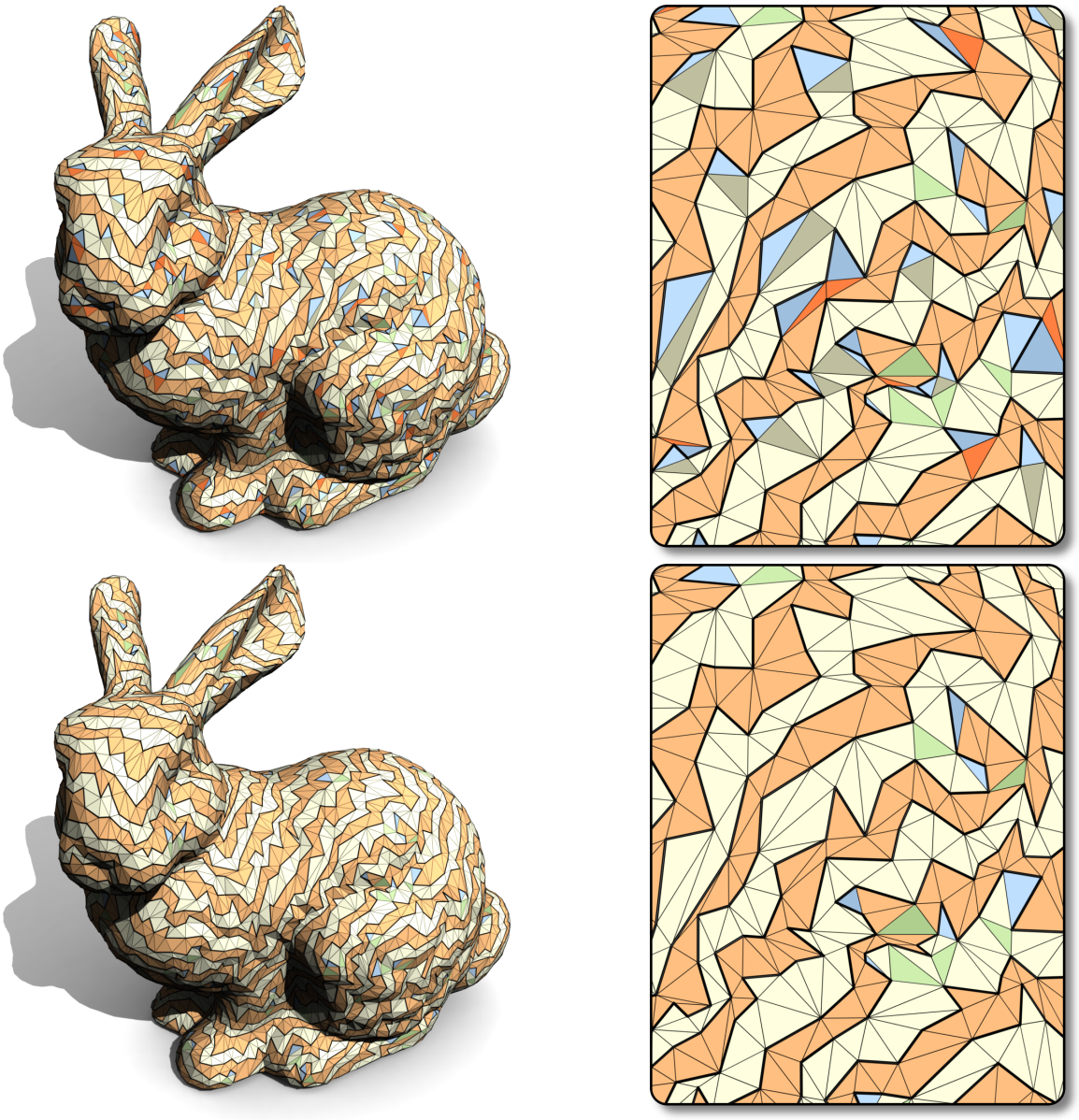


Figure 15: The ring (black loop) delineates two corridors of triangles. Normal T_1 triangles (cream/orange) have one ring edge, dead-end T_2 triangles (blue) have two ring edges, and T_0 triangles (green) comprising bifurcations have no ring edges. Adjacent T_0 (gray/red) and T_2 triangles (top) are represented internally as inexpensive T_1 triangles (bottom), thereby significantly reducing storage.

Omitted V entries for T_1 and T_2 triangles: Most triangles are of type T_1 or T_2. Two of their vertex references (V entries of the CT) which are on the ring are defined implicitly and need not be stored. Thus, we store two references, $L[v]$ and $R[v]$, per ring vertex and assume that triangle $2v$ has vertices $(v, L[v], v.N)$ and triangle $2v+1$ has vertices $(v.N, R[v], v)$, where $v.N = (v+1) \bmod m_r$ is the next vertex after v on the ring, and where m_r denotes the number of ring vertices. Although this data structure has two entries for each T_2 triangle, the cost of this redundancy is amortized, because typically there are far fewer T_2 than T_1 triangles.

Omitted O entries for cheap T_1 and T_2 triangles: We do not store O table entries for the "cheap" T_1 and T_2 triangles that are not adjacent to a T_0, because we can access the opposite corners directly from neighboring ring vertices in constant time.

Ring-expander construction of the ring: We propose a simple (linear time and space) greedy approach for computing a ring that, in all tested cases, either produces a Hamiltonian cycle or leaves a small proportion (only 0.005%) of isolated vertices. Our RING-EXPANDER algorithm tends to minimize the number of T_0 and T_2 triangles.

Wart skipping: To further reduce storage, we conceptually modify the ring to replace **warts**—T_2 triangles adjacent to T_0 triangles— which allows the expensive T_0 triangles adjacent to warts to be represented as cheap T_1 triangles (Fig. 15).

6.2 The LR Representation

In this and the following section, we outline the LR (Laced Ring) approach, describe its representation, and discuss its construction and use. We focus here on a simple representation aimed at minimizing the number of references per triangle. A variation aimed at minimizing the number of bits per triangle is discussed in Section 6.4.

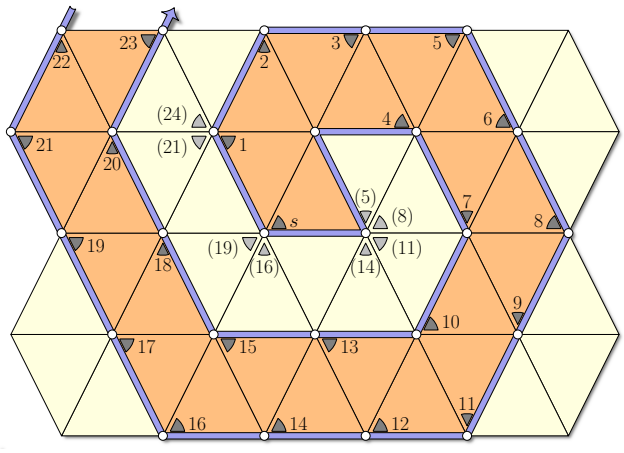


Figure 16: RING-EXPANDER traversal. The corners are numbered in the order in which they are visited, starting with the seed s. Corners of cream triangles that are marked with numbers in parenthesis are corners that are temporarily visited during Ring-expander traversal, but the traversal is backtracked because the vertex incident on the corner has been previously visited.

6.2.1 Topological Domain

We assume that the triangle mesh is a connected manifold without boundary. Meshes with boundaries can be converted to closed manifolds by adding a dummy vertex v and a fan of dummy triangles around v that are joined with the border edges. We discuss the implementation to handle meshes with boundaries further in Section 6.2.6. Non-manifold meshes that represent the boundary of a solid may be converted to pseudo-manifolds while minimizing vertex replication [36], and as such can be represented compactly using our LR data structure.

6.2.2 The Ring

We first select and orient a manifold loop of mesh edges that visits most—and ideally all—vertices. We call it the **ring** and its edges the **ring edges**. The remaining edges are called **transversal**. Assume that the mesh has m vertices, out of which m_r vertices are on the ring. We want to minimize the number $m_i = m - m_r$ of **isolated vertices** that are not on the ring.

The perfect solution, i.e., a Hamiltonian cycle of edges, has been studied in graph

theory. Karp [27] proved that finding the Hamiltonian cycle of a graph is a NP-complete problem. Instead of finding a perfect solution, we propose a linear time approximate solution which produces a ring that is nearly Hamiltonian.

To construct the ring, we use the following greedy RING-EXPANDER algorithm. We begin by marking each triangle t and vertex v as unvisited by setting the flags $t.m$ and $v.m$ to false, respectively. We then pick a random **seed corner** s, from which we perform an invasion that visits most vertices and about half of the triangles. We ensure that the visited region is edge-connected, has no interior vertices (surrounded by only visited triangles), and is bounded by a single manifold loop of edges (i.e., the ring). The RING-EXPANDER code, using corner operators, is simple:

```
c = s;                          // start at the seed corner s
c.n.v.m = c.p.v.m = true;    // mark vertices as visited
do {
    if (!c.v.m) c.v.m = c.t.m = true;      // invade c.t
    else if (!c.t.m) c = c.o;    // go back one triangle
    c = c.r;  // advance to next ring edge on the right
} while (c != s.o);        // until back at the beginning
```

RING-EXPANDER uses corner c to keep track of the current vertex $c.v$ and triangle $c.t$ being considered for invasion. The ring constructed so far separates the invaded triangles (orange) from the other ones (cream); see Fig. 16. If $c.v$ has not been visited, we invade triangle $c.t$, which has the effect of expanding the ring by replacing the ring edge facing c with the other two edges of the invaded triangle. Otherwise, if $c.v$ has been visited, we backtrack until we find a ring edge through which we may continue the invasion. This backtracking is accomplished without a stack or recursion by sliding along the ring ($c = c.o$) and by using the $t.m$ and $v.m$ flags as "breadcrumbs" to keep track of where we have been. These flags are stored efficiently in two vectors of bits. Examples of RING-EXPANDER on the Horse and Bunny meshes are shown in Fig. 17, Fig. 18 and Fig. 19.

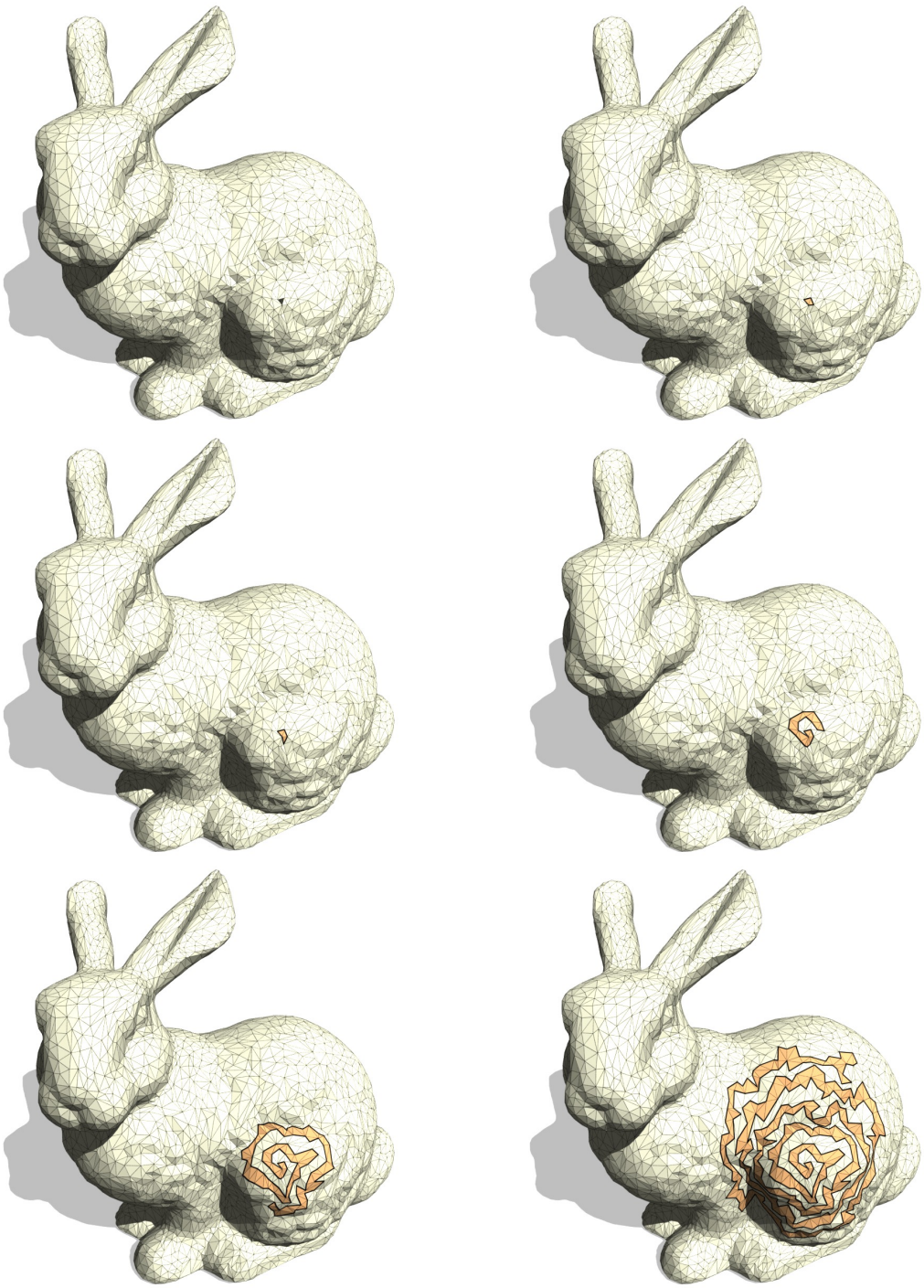


Figure 17: A few initial states of RING-EXPLANDER on the bunny mesh. Orange triangles are the visited ones.

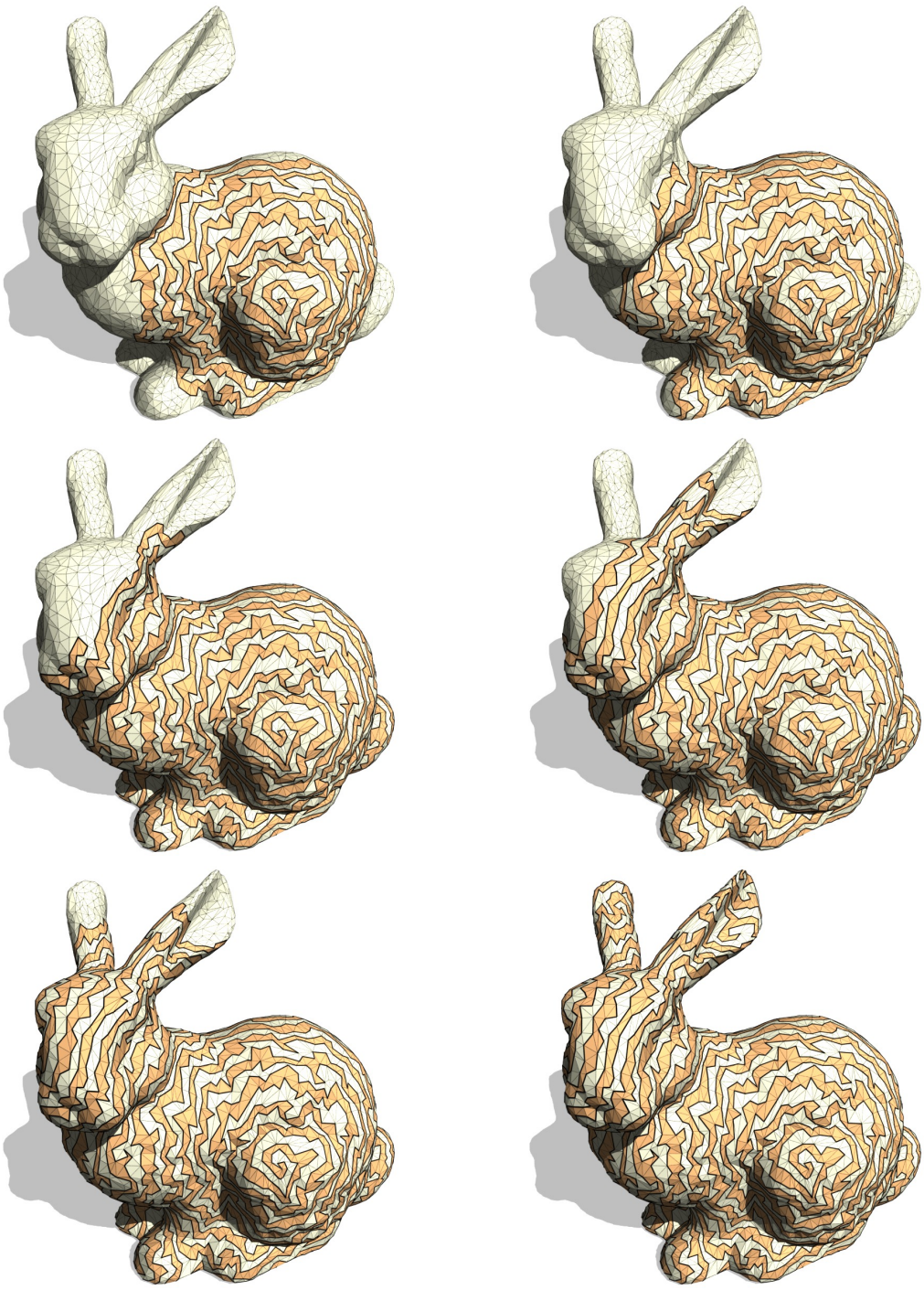


Figure 18: A few final states of Ring-explander on the bunny mesh. Orange triangles are the visited ones.

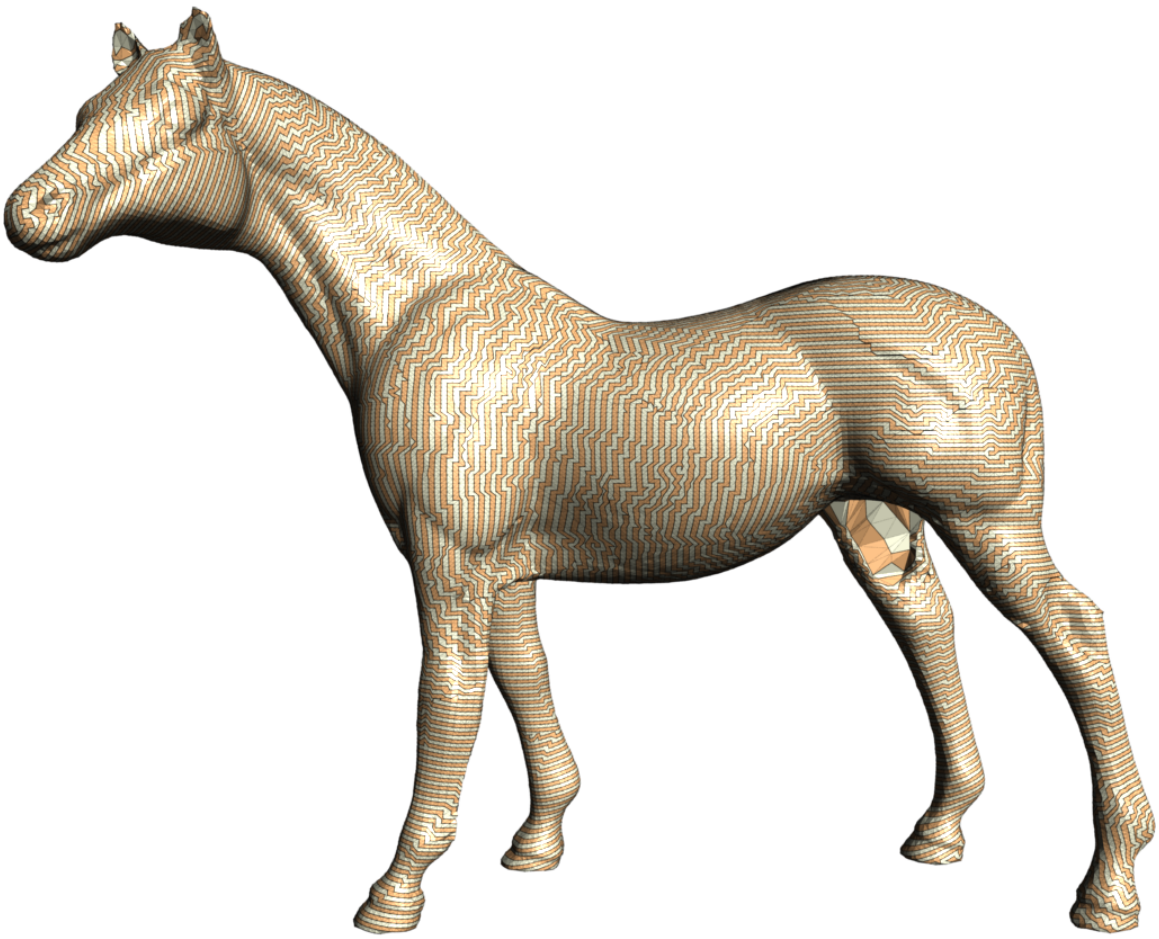


Figure 19: RING-EXPLANDER on the horse mesh. Orange triangles are the visited ones. Note that in the inner thigh, the triangles are larger. This is because during range-scanning, the surface in the inner thigh was not sampled, therefore the hole was filled using a simple hole-filling algorithm.

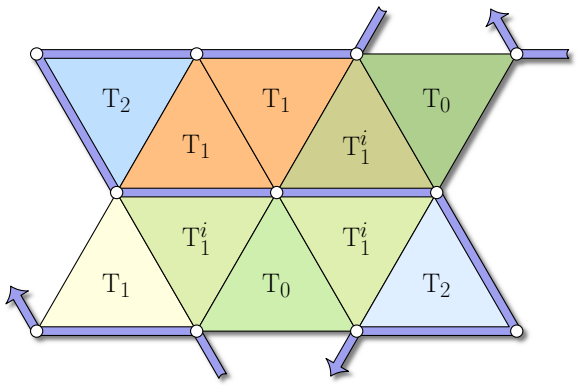


Figure 20: Triangles are classified based on their number of ring edges and whether they are adjacent to a T_0 triangle.

RING-EXPANDER's complexity is linear in space and time, since it visits each corner at most once. The construction is fast, and can process around 30 million triangles per second.

In an attempt to minimize the number of isolated vertices (those not on the ring) m_i, we run RING-EXPANDER several times with random seed corners and retain the seed leading to the smallest m_i, which usually is negligible with respect to m and sometimes is zero. On average, the first run yields a ratio m_i/m of 0.005% averaged over our test models.

6.2.3 Ring-based Classification of Triangles

To simplify exposition, we distinguish several kinds of triangles (see Fig. 20). T_0 triangles (bifurcations) have no ring edges; T_1 triangles (the most common kind) have exactly one ring edge each; T_2 triangles (dead-ends of the "corridors") have two edges on the ring. T_1^i and T_2^i are "expensive," **irregular** T_1 and T_2 triangles that share an edge with at least one T_0 triangle. Finally, we call a T_2 triangle that is adjacent to a T_0 triangle a **wart**. Such pairs of triangles are denoted T_0^w and T_2^w.

A triangle incident upon an isolated vertex must be T_0 because, clearly a T_2 triangle cannot have an isolated vertex, since all of its three vertices are on the ring. Furthermore, a T_1 triangle has two consecutive ring vertices, v and $v.N$. If its third

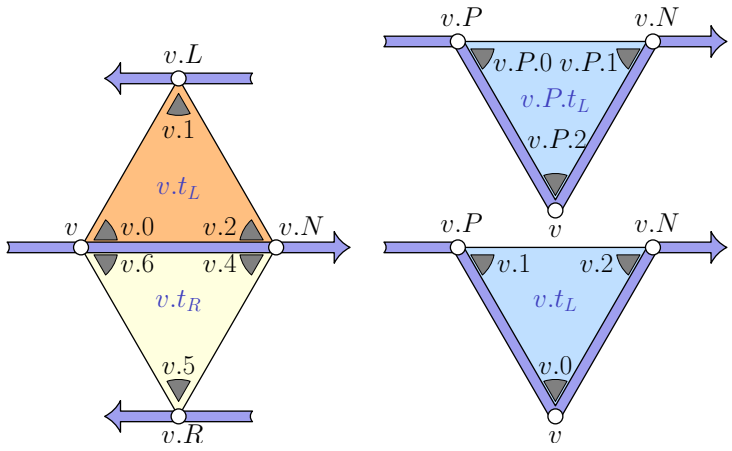


Figure 21: Left: Left and right triangles $v.t_L$ and $v.t_R$ are defined for each ring edge $(v, v.N)$. Their corners are labeled $(v.0, v.1, v.2)$ and $(v.4, v.5, v.6)$. Right: Redundant (top) and canonical (bottom) representation of a T_2 triangle.

vertex w were isolated, then our construction algorithm would have included w in the ring between v and $v.N$, turning the triangle into a T_2.

6.2.4 Representing Incidence

We identify the ring vertices by integers between 0 and $m_r - 1$ assigned in order of appearance along the ring (starting from an arbitrary vertex). Hence, the references $v.P$ and $v.N$ to the vertices that respectively precede and follow v on the ring may be computed as $v.P = (v + m_r - 1) \bmod m_r$ and $v.N = (v + 1) \bmod m_r$. Vertices with indices between m_r and $m - 1$ are isolated vertices.

Each edge $e = (v, v.N)$ of the ring is associated with a starting vertex v and with two incident triangles: $v.t_L$ on the "left" and $v.t_R$ on the "right." We renumber the triangles so that $v.t_L = 2v$ and $v.t_R = 2v + 1$. Triangle $v.t_L$ has vertices $(v, v.L, v.N)$, where $v.L$ is stored in the L table as $L[v]$. Similarly, triangle $v.t_R$ has vertices $(v.N, v.R, v)$, where $v.R$ is stored in the R table as $R[v]$; see Fig. 21. T_0 triangles, which have no ring edges, are not stored in the LR table. Rather, they are represented using the regular Corner Table (arrays V and O, or simply VO), and are assigned indices $2m_r$ and above.

We call the corners of the $v.t_L$ and $v.t_R$ triangles incident upon a ring edge the **ring corners**. We label them $v.0$, $v.1$, $v.2$, $v.4$, $v.5$, and $v.6$, as shown in Fig. 21, and assign to corner $v.i$ the integer index $8v + i$. Thus the **offset** i of a corner c is determined by the three least significant bits of c. By shifting the base of this scheme by eight rather than six for each vertex, we are not using corner IDs $8v + 3$ and $8v + 7$. This irregular assignment of indices speeds up some of the corner operators by allowing bit shifts and masks to be used in place of division and modulo. Although corners $8v + 3$ and $8v + 7$ do not exist as was the case with SQuad, no storage is wasted on these unused indices, since we do not allocate any space to a corner. Not using consecutive corner numbers limits the size of the mesh that can be stored, but using 32-bit references to opposite corners eliminates this concern for all practical purposes.

Note that there are two possible representations for the corners of a T_2 triangle: it could be associated with both the first and second ring edges. For many traversal operations this is not a problem, but when unique corner references are desired, our convention is to associate the T_2 triangle with its second ring edge. We say that the other three corner references (associated with the first ring edge) are **redundant**. We can easily detect that a reference is redundant and convert it to the corresponding canonical reference. For example, given a corner $c = v.P.2$ (see Fig. 21, top right), we detect that c is redundant because $v.P.L = v.N$, and compute the canonical corner reference as $v.0$. Mappings of other corners, $c = v.P.0$ and $c = v.P.1$, and corners in the symmetric configuration are handled similarly.

When performing mesh traversals, a corner operator that is useful is the $v.c$ corner operator. In LR, we can obtain a reference $v.c$ to a corner of a given ring vertex v as $v.c = v.0 = 8v$ and visit the triangles incident on v using the $c.s$ operator. A reference to one corner of each isolated vertex is stored explicitly in an auxiliary array C.

If all the vertices were on the ring and if all the triangles were incident upon at

least one edge of the ring, this representation would suffice to support all the standard corner operators, and would store only two references per vertex, or **1 rpt** (since there are roughly twice as many triangles as vertices).

6.2.5 Representing Adjacency

Triangle adjacency is provided by the opposite corner operator $c.o$. Within a **quad** formed by triangles $v.t_L$ and $v.t_R$, $v.1$ and $v.5$ are opposite corners. Hence $v.1.o$ and $v.5.o$ are defined implicitly, can be obtained trivially, and need not be stored. In a T_2 triangle, $v.2.o$ and $v.4.o$ may be obtained by first remapping $v.2$ and $v.4$ to their redundant counterparts $v.P.1$ and $v.P.5$, and then computing their in-quad opposites (see Fig. 21 and Fig. 22).

Opposite corners of T_1 and T_2 triangles that do not lie in a T_0 can be obtained for T_1 and T_2 corners $v.0$, $v.2$, $v.4$, and $v.6$ by visiting nearby vertices on the ring. That is, when crossing a transversal edge via $c.o$, one or both of the other edges in the adjacent T_1 or T_2 triangle must be ring edges. For instance, if $v.N.L = v.L$, then $v.0.o = v.N.2$. Otherwise, $v.L.P.L = v.N$ and $v.0.o = v.L.P.0$; see Fig. 22. When $c.o$ lies in a T_2 triangle, we must also remap the corner if it is redundant.

If $c.o$ lies in a T_0, on the other hand, we cannot reach it via the ring so we store in L or R a bit signaling that $c.t$ is an expensive T_1^i or T_2^i triangle. In this case, rather than storing $v.L$, we store in $L[v]$ an index into a condensed corner table VO^* (and similarly for $v.R$). VO^* holds triplets $(v.L, v.0.o, v.2.o)$ and $(v.R, v.4.o, v.6.o)$.

Finally, to determine the opposite corners for corners c in T_0 triangles, we consult the O table, which holds opposites for all three corners of such triangles.

6.2.6 Meshes with Borders

As discussed previously, LR can handle meshes with boundary by introducing triangles that join boundary edges to a single dummy vertex v. If there are several boundary loops, this addition creates a non-manifold vertex. We ensure that v is not

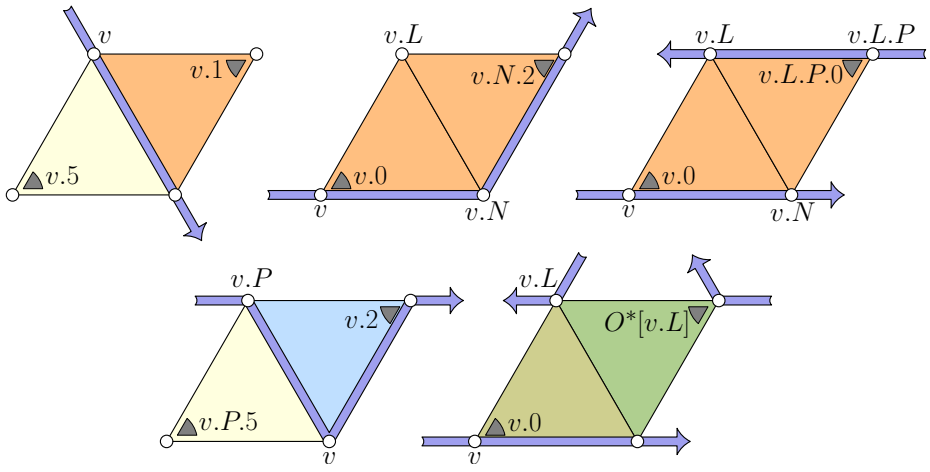


Figure 22: Different cases for computing $c.o$. The cases are (left-to-right, top-down): (i) if $v.t_L$ and $v.t_R$ are T_1 triangles, then $v.1.o = v.5$, (ii) if $v.L == v.N.L$, then $v.0.o = v.N.2$, (iii) if $v.L.P.L == v.N$, then $v.0.o = v.L.P.0$, (iv) if $v.t_L$ is a T_2 triangle, then $v.2.o = v.P.5$, and (v) if $v.t_L$ is a T^i triangle, then $v.0.o = O^*[v.L]$. The formulae for opposites for other cases can be derived by symmetry, e.g. in case (i), $v.5.o = v.1$, in case (ii), if $v.R == v.N.R$, then $v.6.o = v.N.4$, etc.

a part of the ring by initially marking it as visited, which guarantees that we never invade any of the dummy triangles incident on v. Any reference to v in the LR table is replaced with a special **null** index.

6.2.7 Implementation of Operators

We summarize here the implementation of the standard corner operators for LR.

- **c.v**: If $c \geq 8m_r$, then $c.t$ is a T_0 triangle and $c.v = V[i]$, where $i = c - \lfloor c/4 \rfloor - 6m_r$. (This subtraction of $\lfloor c/4 \rfloor$ restores the base to six to avoid unused corners in the VO table.) Else, we compute $v = \lfloor c/8 \rfloor$ and use the relative corner offset $c \bmod 8$ to select among v, $v.N$, $v.L$, and $v.R$ (see Fig. 21).

- **c.o**: If $c \geq 8m_r$, then $c.t$ is a T_0 triangle and $c.o = O[i]$, where $i = c - \lfloor c/4 \rfloor - 6m_r$. Else, if $c < 8m_r$, we let $v = \lfloor c/8 \rfloor$ and distinguish five cases (Fig. 22). In the first case, $v.1.o = 8v + 5$ and $v.5.o = 8v + 1$. In the next three cases, we infer the opposite from the L and R tables and ring vertices. For example, if $v.L = v.N.L$, then $v.0.o = 8v.N + 2$. In the fifth case, when c is in a T^i triangle,

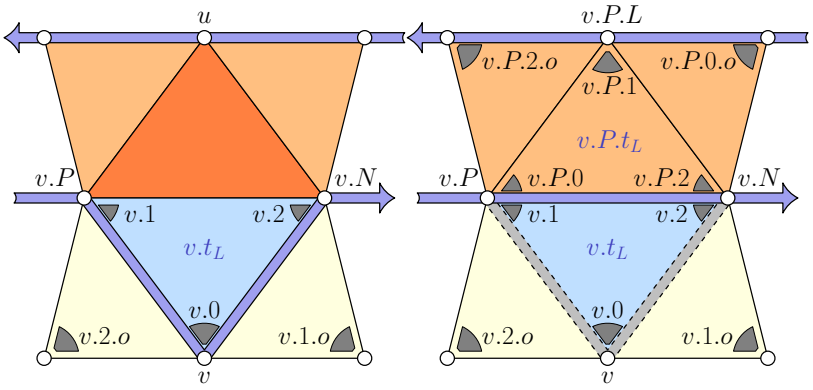


Figure 23: Wart skipping treats T_0 triangles (red) adjacent to T_2 warts (blue) as T_1 triangles (cream/orange) by excluding the wart from the ring. The T_0 is stored as the first redundant copy of the T_2.

we look up $c.o$ using $v.L$ or $v.R$ as an index into the VO^* table. The formulae for the other cases can be derived by symmetry.

- **v.c**: If $v \geq m_r$, then v is isolated and $v.c = C[v - m_r]$. Else if $v < m_r$ then v is a vertex on the ring. In such cases, if $v.L = v.N.N$ (redundant T_2 triangle), then $v.c = 8v.N + 1$ (and similarly for $v.R$); otherwise $v.c = 8v$.

- **c.t**: The triangle $c.t$ of corner c is defined as $\lfloor c/4 \rfloor$.

- **t.c**: The first corner $t.c$ of triangle t is defined as $4t$.

- **c.n**: The next operator is defined as $c.n = c - 2$ if $c \bmod 4 = 2$; otherwise $c.n = c + 1$.

- **c.p**, **c.s**, **c.l**, and **c.r** are derived from the operators discussed above (see Section 1.2).

6.3 Wart Skipping

The number of T_0 triangles is typically small compared to the number of T_1 triangles. However, the connectivity information associated with a T_0 triangle requires significantly more storage, both for itself and for its adjacent triangles: 6 references to

represent each T_0 triangle, and 3 references for each of the three T^i triangles adjacent to the T_0 triangle, therefore totaling 15 references. Hence, it is important to reduce the number of T_0 triangles. To do so during construction of LR, we identify warts: T_2 triangles that are adjacent to T_0 triangles. (When more than one T_2 triangle is adjacent to a T_0, an arbitrary one is chosen as the wart.) Because each T_2 triangle is duplicated, we may reclaim the storage for the redundant copy of the T_2 triangle and use it to represent the T_0. That is, for a T_2 triangle $(v.N, v, v.P)$ adjacent to a T_0 triangle $(v.P, u, v.N)$, we store u rather than $v.N$ in $L[v.P]$ (see Fig. 23). We also store a bit in the entry for the T_0 to indicate that it has been paired with a wart, and use T_0^w to denote such triangles. Warts are denoted T_2^w.

To correctly process T_0^w and T_2^w triangles, we conceptually modify the ring by skipping over the wart and its tip vertex v when accessing the T_0^w, which in effect makes $(v.P, v.N)$ a ring edge and turns the T_0^w triangle into a regular T_1 (Fig. 23). This reclassification of the T_0 triangle also affects all incident T_1^i or T_2^i triangles, which unless they are adjacent to another T_0 now become regular (cheap) triangles.

The negative impact of wart skipping on the performance of the corner operators is small: For $c.v$ and $c.o$, we let $v.P.N = v.N$ and $v.N.P = v.P$ whenever accessing a T_0^w triangle. Opposites of wart tip corners are also redefined as $v.0.o = v.P.1$ and $v.6.o = v.P.5$, and conversely for T_0^w tip corners. Aside from this change, the corner operators for warts stay the same.

Wart skipping reduces the storage cost by as much as 15 references per skipped wart: we make actual use of the redundant reference for the T_2 triangle (described above), reduce the 6-reference cost for the T_0 triangle to a single entry, and reduce the 4-reference cost of all adjacent T_1^i and T_2^i triangles to a single entry. In practice, because T_0 and T_2 triangles often come in pairs, wart skipping usually reduces the number of T_0 triangles by more than half.

6.4 Bit-Efficient LR Representation (BELR)

The LR representation discussed so far has been optimized to reduce the number of integer references per triangle. Its storage efficiency can be improved by carefully considering how these references are encoded. In particular, by changing the traversal of RING-EXPANDER to produce a ring with greater locality of reference, we allow short relative indices to be used even for very large meshes, though possibly using a larger *number* of references. This space-optimized representation, the Bit-Efficient LR (BELR), is discussed below.

6.4.1 Relative Indexing

The LR table, as presented above, stores 32-bit integer references to vertices. In practice the index difference, or offset, between a ring vertex v and its left and right neighbors $v.L$ and $v.R$ is often small enough to fit in 16 bits, even when the mesh has far more than 2^{16} vertices. We exploit this observation and store the offsets $v.L - v$ and $v.R - v$ (modulo the number of ring vertices m_r) instead of the absolute indices. This saves us storage space as absolute indices are stored as 32-bit references, while the offsets are stored as 16-bit references. For large meshes, however, the depth-first traversal of RING-EXPANDER often results in very long triangle strip corridors between bifurcations (Fig. 24, top). In general, more bifurcations, and thus shorter corridors, lead to smaller offsets.

A breadth-first strategy for RING-EXPANDER (Fig. 24, bottom) generates shorter offsets, but favors bifurcations—i.e. expensive T_0 triangles—over long corridors—i.e. cheap T_1 triangles. Hence, we advocate a compromise (Fig. 24, middle): A hybrid breadth- and depth-first traversal that balances the number of bifurcations and the magnitudes of offsets. It modifies RING-EXPANDER to interrupt the depth-first traversal every k steps and resets the traversal using breadth-first backtracking. Setting, $k = 1$ results in a pure breadth-first traversal, while $k = \infty$ yields a depth-first

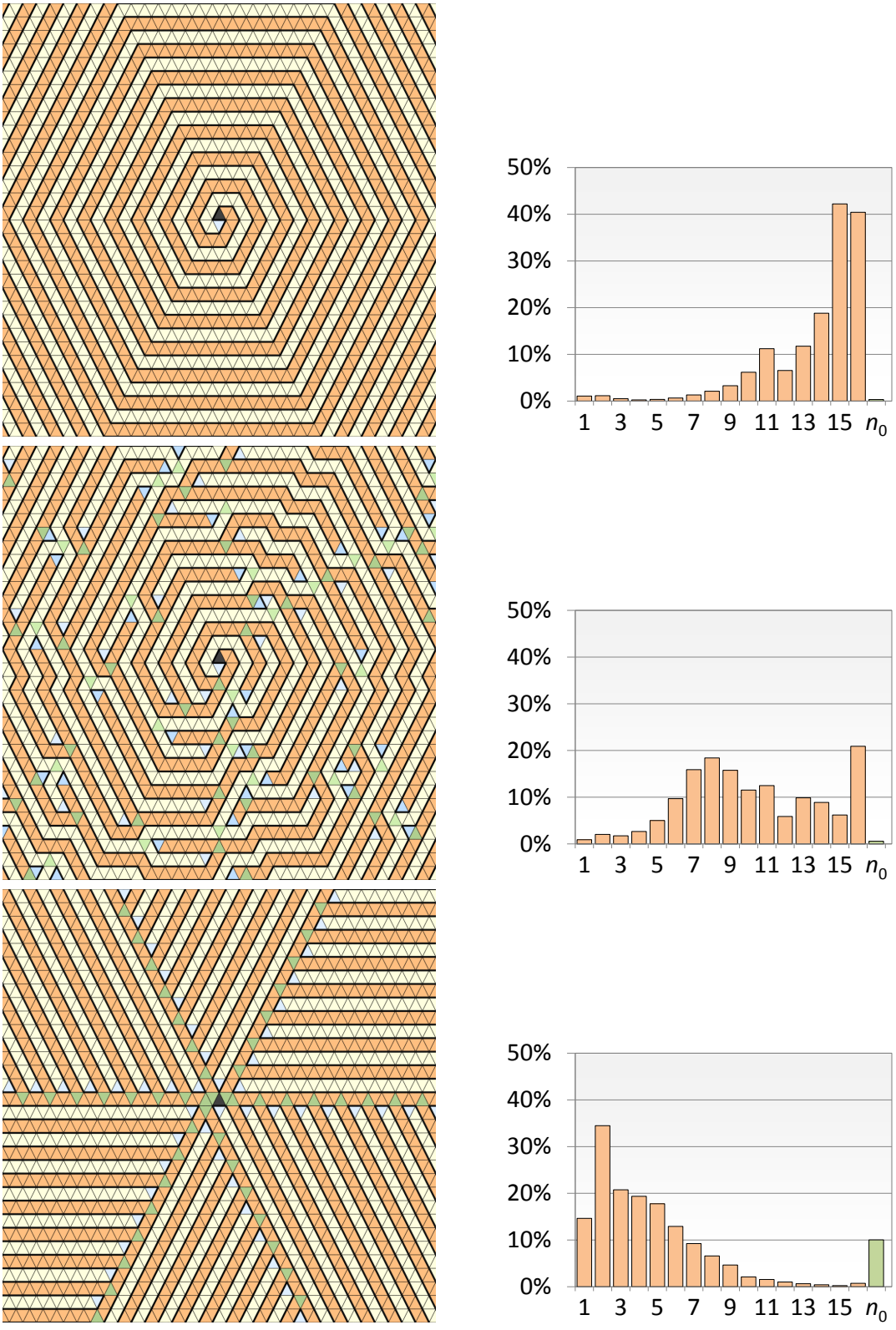


Figure 24: Depth-first (top), hybrid $k = 32$ (middle), and breadth-first (bottom) traversals, with offset distributions in number of significant bits (1 to 16) and fractions of T_0 triangles (green, rightmost column), which are 0.33%, 0.56%, and 10%.

traversal. Intermediate values of k may be used to tune the number of bifurcations and distribution of offsets.

Our HYBRID-RING-EXPANDER algorithm records backtracking corners in a double-ended queue `d` that is initially empty:

```
c.n.v.m = c.p.v.m = true;      // mark vertices as visited
while (true) {
    if (!c.v.m) {              // has c.v been visited?
        c.v.m = c.t.m = true; // invade c.t
        d.push_back(c.l);      // push left and right...
        d.push_back(c.r);      // ... neighbors onto deque
        n++;                   // increment triangle count
    }
    if (d.empty()) break;
    if (n % k == 0) c = d.pop_front();   // breadth-first
    else            c = d.pop_back();    // depth-first
}
```

Though slightly more complex than RING-EXPANDER, this hybrid method still achieves a throughput of 25 million triangles/second.

Rings generated with an optimal value of k tend to have offsets that can be stored as 15-bit signed integers. When this is the case for both of a pair of $v.t_L$ and $v.t_R$ triangles, we store the offsets as 16-bit entries in the LR table. LR entries that require more bits are handled using one level of indirection into the VO^* table, which is indexed by combining bits from the L and the R entries into a 26-bit reference a. VO^* stores the corresponding $v.L$ and $v.R$ indices in consecutive locations $VO^*[a]$ and $VO^*[a+1]$ using 32 bits each. We use T^l to identify triangles that require this extra level of indirection and specification of $v.L$ or $v.R$ using **long references**. As in standard LR, T^i denotes irregular triangles adjacent to a T_0 that also require long indexing into VO^*, for which one vertex and two opposite corners are stored.

One may think of the VO^* table as a full corner table, where references that

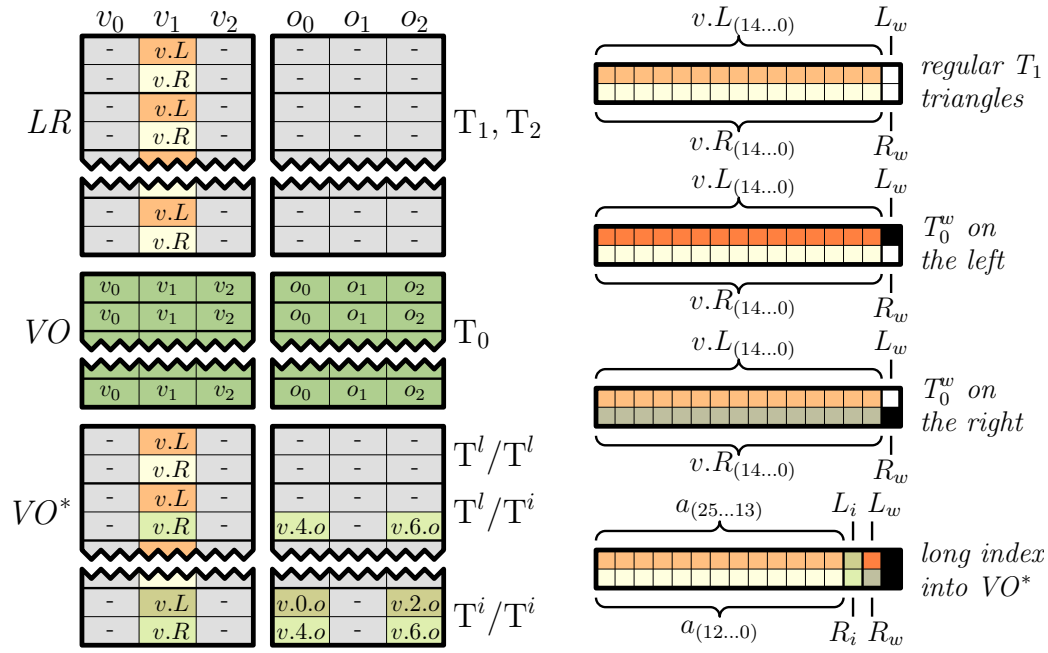


Figure 25: Bit-Efficient LR storage. Left: Each row corresponds to a triangle. Gray shaded vertices and corners are implicit and are not stored. Right: Encoding of LR table using 16 bits per reference.

are already known are not stored (see Fig. 25). Our implementation discards the unused VO^* entries and packs this table into a single linear array of integer references. Because we always arrive at a sequence of entries in this table knowing the type of each triangle in a pair—T^l/T^l, T^l/T^i, T^i/T^l, or T^i/T^i—there is no ambiguity what the next 2, 4, or 6 integer entries represent. In particular, the first two references of a tuple always store $v.L$ and $v.R$.

6.4.2 Storage Format

For each LR entry we store two bits, L_w and R_w, identifying one of four configurations: (1) a pair of T_1 triangles, (2) a T_0^w on the left or (3) a T_0^w on the right, or (4) a pair of T^l or T^i triangles that require long indexing. Note that T_0^w triangles can appear on the left or right, but not both simultaneously, as the triangle paired with the T_0^w triangle is adjacent to a T_2^w triangle and has at least one ring edge. Thus, the two bits stored in the LR table indicate whether to skip warts on the left and on the

right, with the unused double-wart combination signaling the need for a long index (see Fig. 25).

As discussed above, when necessary, we combine the LR entries into a 26-bit index a into the VO^* table. With two additional bits out of the 32 already used, the remaining four bits are used to encode left and right wart skips (since a triangle may require a long index into the VO^* table *and* a wart skip) and whether $v.t_L$ and/or $v.t_R$ is adjacent to a T_0, i.e. if it is an irregular T^i.

The VO table stores first a list of all T_0 triangles as six references per triangle. Any subsequent vertices and opposite corners that cannot be represented directly in the LR table are stored as variable-length records, in no particular order, in the VO^* table. The index a and the combination of L_i and R_i bits, which distinguish T from T^i triangles, are sufficient to determine the record type.

CHAPTER VII

ZIPPER

7.1 Introduction

In this chapter, we discuss Zipper. We start with a brief overview. Zipper uses on average only 5.98 bits per triangle (*bpt*) which represents a 34.8x improvement over ECT, a 5.8x improvement over LR and a 4.4x improvement over BELR. Its data structure can be constructed in linear space and time from a standard incidence format, and supports all standard random-access and mesh traversal operators in constant time.

Zipper uses the Ring-Expander algorithm from LR to build a ring and to renumber the vertices, triangles and corners. Zipper provides three improvements over LR:

1. Zipper avoids storing most of the $v.L$ and $v.R$ references explicitly. Instead it stores a pair of 3-bit codes for most ring vertices. These identify wart triangles and encode deltas rather than absolute references. To help resolve these references in constant time, Zipper stores two additional bits (amortized) per triangle.

2. To reduce the number of T_0 triangles, Zipper applies the Ring-Bender algorithm, as described in Section 7.5.

3. Zipper reduces by 2.5x the storage cost associated with T_0 triangles. It does so by inferring connectivity by locally traversing a portion of the ring.

We propose a novel coding: we store $v.P.L - v.L$ and $v.P.R - v.R$ (described in Section 7.2). This differs from the $v.L - v$ and $v.R - v$ coding we use in BELR.

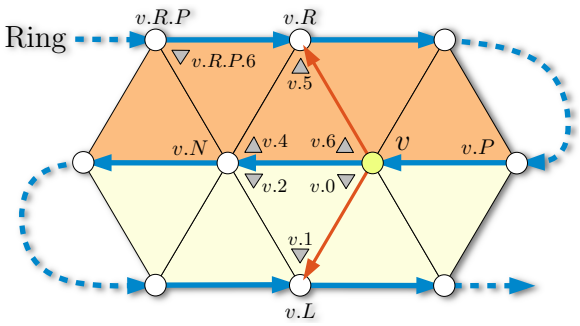


Figure 26: With each ring vertex v, LR stores references $v.L$ and $v.R$ to the tips of the two triangles incident upon ring edge $(v, v.N)$.

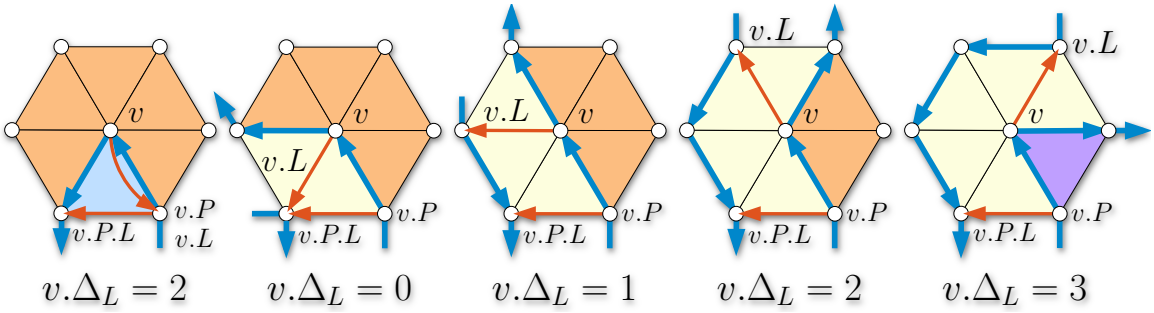


Figure 27: The ring (blue) passes through a valence-six vertex v. The $v.L$ and $v.P.L$ references are shown as red arrows. In absence of incident T_0 triangles, only deltas in $\{0, 1, 2, 3\}$ are possible at v.

This discovery of a new differential coding is a major factor in the large reduction in storage cost offered by Zipper over LR.

7.2 Delta codes for vertices

The most significant storage improvement in Zipper comes from the observation that, in most cases, $v.L$ can be recovered by subtracting a small integer from $v.P.L$ (and similarly for $v.R$). Our tests indicate that in a typical mesh, 95% of the **deltas** $v.\Delta_L = v.P.L - v.L$ and $v.\Delta_R = v.P.R - v.R$ are in the set $D = \{0, 1, 2, 3\}$. Hence, in these cases, Zipper stores only two bits per delta, instead of the 32-bit $v.L$ or $v.R$ references. Note that these delta values are typically non-negative, because the ring bounds "strips" of edge-adjacent triangles, and therefore $v.L$ and $v.R$ decrease as v increases (see Fig. 26).

To understand why most deltas are between 0 and 3, consider the common case of a valence-6 vertex, as shown in Fig. 27, where the portion of the ring from $v.L$ to $v.P.L$ is on the link of v. When the triangles incident on v are T_1 or T_2 (such triangles make up over 96% of the mesh), only the $0, 1, 2, 3$ deltas are possible. This observation also holds for vertices of valence lower than 6.

Configurations where the delta is not in D are flagged as **exceptions**. For each exception we store a full 32-bit reference to the corresponding tip vertex. We refer to such tip vertices ($v.L$ or $v.R$) as **key vertices**.

As in LR, we identify warts (pairings of a T_0 triangle with an adjacent T_2 triangle). Because T_2 triangles have two ring edges and would thus be represented twice, we store the adjacent T_0 in place of the first (along the ring) copy of T_2. Such a wart pair, which we label T_0^w/T_2^w, is illustrated by triangles #95 and #97 in Fig. 31. Because not all T_2 triangles are adjacent to a T_0, we store a **wart bit** with each ring triangle to indicate whether it is a T_0^w.

Unlike in LR, in Zipper, we use a special encoding of T_2^w triangles $(v.P, v, v.N)$ to reduce the number of exceptions. Rather than storing $v.P$ (vertex #47 in Fig. 31) as the T_2^w tip vertex, which often would incur an exception, we set delta to zero and rely on the fact that we can always recover the tip vertex $v.P$ from v when the wart bit of the previous triangle is set. This encoding also ensures that the delta of the following triangle (for example triangle #99 in Fig. 31) is computed with respect to the T_0^w tip vertex (vertex #37 in Fig. 31). Furthermore, it avoids having to encode a second exception.

Adding a wart bit to the two-bit deltas results in a 3-bit encoding of each triangle.

We reserve the 3-bit pattern 111 to mark exceptions, which would otherwise correspond to the rare case of $\Delta = 3$ in a T_0^w wart triangle.

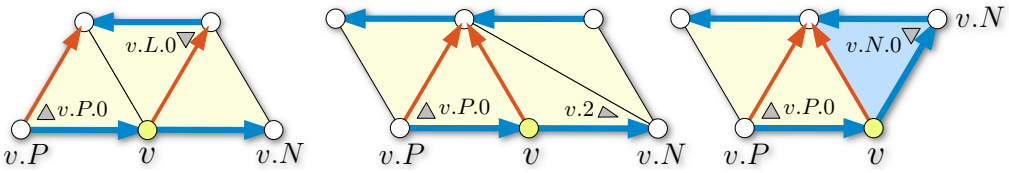
7.3 Blocks

To lower the cost of recovering $v.L$ and $v.R$ without summing all preceding deltas along the ring, we force an exception every 32 $v.L$ and $v.R$ references, and store these explicitly. We refer to such a sequence of 32 references as a **block**. Thus, computing $v.L$ or $v.R$ requires summing at most 31 deltas. To accelerate this key step, we have devised an efficient technique that computes the sum of deltas using bit-level operations, without executing a loop, as described in Section 7.6. Because the number of exceptions per block varies, each block stores a single index into an **exception table** (a dense array of key vertex references). The storage required for a block includes (1) a 32-bit reference for the first vertex of each block, (2) a sequence of 32 3-bit delta/wart codes, and (3) a 32-bit pointer into the exception table, resulting in a minimum of 160 bits per block (5 *bpt*). We chose a block size of 32 as a compromise: (1) we want the block to be large, so that we can amortize the cost of storing the $v.L$ and $v.R$ references at the beginning of each block and (2) we want to be able to compute the sum of the deltas efficiently. Exploring the use of blocks of 64 has been left for future work.

Whereas in LR each reference is stored as 32 bits, Zipper allows references to be stored in only 5 bits. References that generate exceptions require 36 bits. Consequently, in the best case we improve storage by 6.4x over LR, and in the worst case, when every reference is an exception, we introduce an overhead of $\frac{1}{8}$.

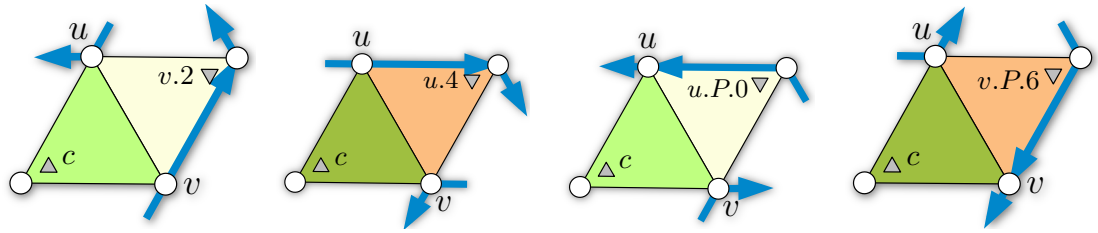
7.4 Computing opposites

LR derives adjacency information (the *c.o* references) by using reference-equality tests and combinations of $v.N$, $v.P$, $v.L$, and $v.R$ references. We explain here how we modified this approach to accommodate Zipper's more compact representation.



(a) $v.\Delta_L = 1$ (b) $v.\Delta_L = 0$

Figure 28: Opposites can be efficiently computed for $\Delta \in \{0, 1\}$. The cases are: (left) if $v.\Delta_L == 1$, then $v.P.0.o = v.L.0$, (middle) if $v.\Delta_L == 0$ and $v.t_L$ is a T_1 triangle, then $v.P.0.o = v.2$, and (right) if $v.\Delta_L == 0$ and $v.t_L$ is a T_2 triangle, then $v.P.0.o = v.N.0$.



(a) $v.L = u$ (b) $u.R = v$ (c) $u.P.L = v$ (d) $v.P.R = u$

Figure 29: The four cases for finding $c.o$ when $c.t$ is a T_0 triangle and $c.o.t$ is a T_1 or T_2 triangle. Given u and v (two of the vertex indices that are stored for T_0 triangles), the four cases are: (a) if $v.L == u$, then $c.o = v.2$, (b) if $u.R == v$, then $c.o = u.4$, (c) if $u.P.L == v$, then $c.o = u.P.0$, and (d) if $v.P.R == u$, then $c.o = v.P.6$.

7.4.1 From T_1 to T_1

For T_1 and T_2 triangles, we do not store opposite corners $c.o$ explicitly, but compute them when needed by decoding the $v.L$ and $v.R$ references, and by using the implicit next $v.N = v+1 \bmod m_r$ and previous $v.P = v-1 \bmod m_r$ ring operators (where m_r is the number of ring vertices). An example of computing opposite corners is shown for $v.6$ in Fig. 26.

When only T_1 and T_2 triangles are involved, we detect the local ring's configuration and return the appropriate corner. The four cases that need to be checked are detailed in LR (see Section 6.2.5). We can often compute opposite corners by examining only the delta, i.e. without fully decoding $v.L$ or $v.R$. As shown in Fig. 28, this is possible whenever $\Delta \in \{0, 1\}$.

7.4.2 From T_0 to T_1

Unlike in LR, which stores both vertices and opposites explicitly for all T_0 triangles, Zipper uses a different procedure for inferring opposites of T_0 corners, when those do not lie in another T_0, and stores only the three vertex references for each T_0 triangle. Given the two vertices of the T_0 not incident on c (see Fig. 29), we use the $v.P$, $v.L$, and $v.R$ operators to navigate to the opposite corner. The four possible configurations are illustrated in Fig. 29.

Because in practice roughly half of the T_0 triangles are adjacent to three T_1 triangles, this elimination of three references per T_0 has a significant impact on resulting storage.

LR stores up to 15 references for each T_0 triangle (3 to its vertices, 3 to opposite corners, and 9 from adjacent triangles). Since, in a typical mesh, the percentage of T_0 triangles varies from 0.5% to 2.0%, using the LR approach to represent T_0 triangles adds between 2.5 and 10 bpt (a cost obtained by amortizing the storage cost of these exceptions over all triangles), and hence dominates the storage cost. With our improved scheme, we store an average of 6.0 references per T_0 triangle.

7.4.3 Hashing to a T_0

When the opposite corner $c.o$ lies in a T_0, it is not possible to find it using only $v.L$, $v.R$, $v.N$, and $v.P$. When c lies in a T_1 (referred to as a T_1^i triangle), LR stores a special pointer into a table that holds both the tip vertex and up to two unknown opposites of the triangle $c.t$. To avoid the cost of storing these exceptional references explicitly, in Zipper, we use an alternative data structure that stores only opposite corners for exceptional configurations. This data structure is used for all opposites that cannot be inferred from the rules above, i.e. for all corners $c.o$ that lie in a T_0.

For space efficiency, we use a d-ary cuckoo hash [18], which maps each key to one of d possible locations, out of which one is guaranteed to hold the hashed item. Aside

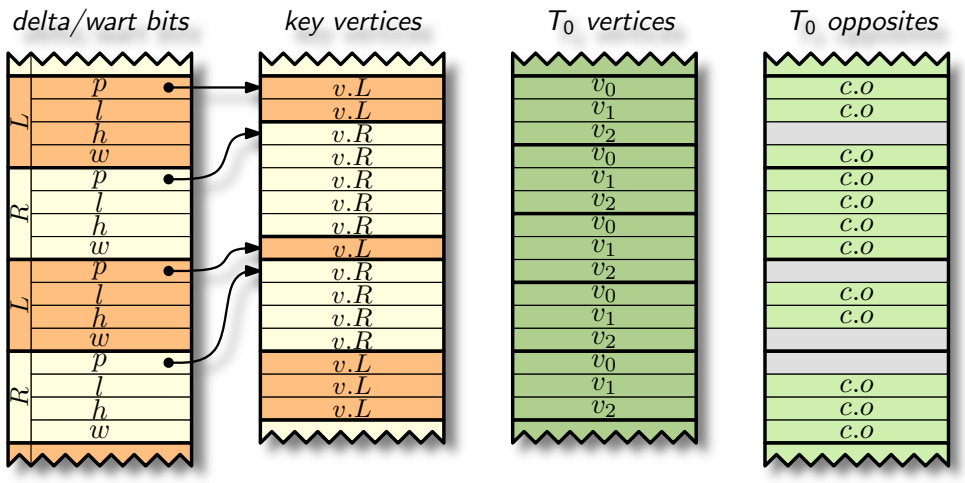


Figure 30: Zipper storage. Delta/wart bits: For each run of 32 $v.L$ or $v.R$ references, we store a block consisting of four 32-bit integers that represent a key vertex pointer p and, for each vertex in the run, a low and high delta bit l and h, and a wart bit w, respectively. Key vertices: Pointer p points to the exception table containing the key vertices. T_0 vertices: vertex IDs for T_0 triangles. T_0 opposites: 4-ary cuckoo hash table containing opposite corner IDs for T_0 triangles.

from $O(N)$ insertion of N items and $O(1)$ lookups, cuckoo hashes have a desirable property in that, as d grows, the maximum allowable load factor f approaches one. In practice, $f = 97\%$ when $d = 4$ (the setting we used), thus only 3% is wasted on empty slots.

In Zipper, we use c as the key and store only the value $c.o$ in the hash. A typical hash lookup generates d possible candidates, which direct us to triangles $c.o.t$ in the T_0 table. Among the d candidates, we identify the one that contains the edge $e = (c.p.v, c.n.v)$ shared with $c.t$. If no such triangle is found, $c.o$ does not exist, indicating that e is a border edge. The cost of storing an explicit opposite reference is thus $\frac{32}{f}$, or about 33 bits when $d = 4$.

One attractive property of Zipper is that, unlike LR, it fully separates the representation of vertices and opposite corners. For those applications that do not require adjacency (e.g. rendering, transmission, etc.), the corner hash may be discarded without having to modify the Zipper incidence representation.

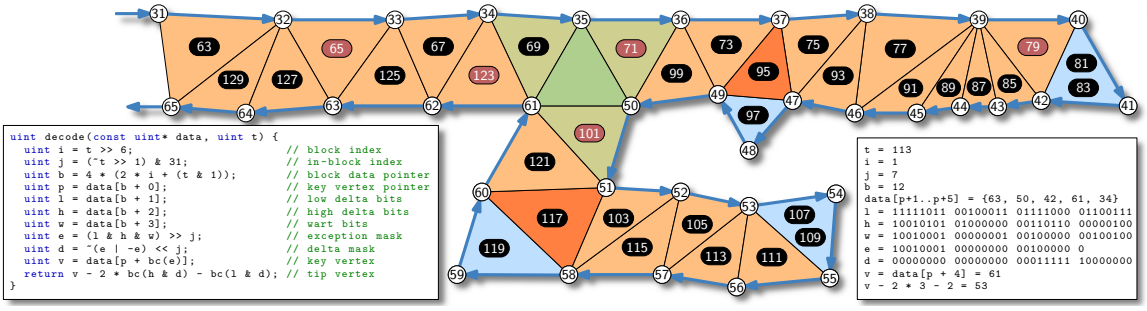


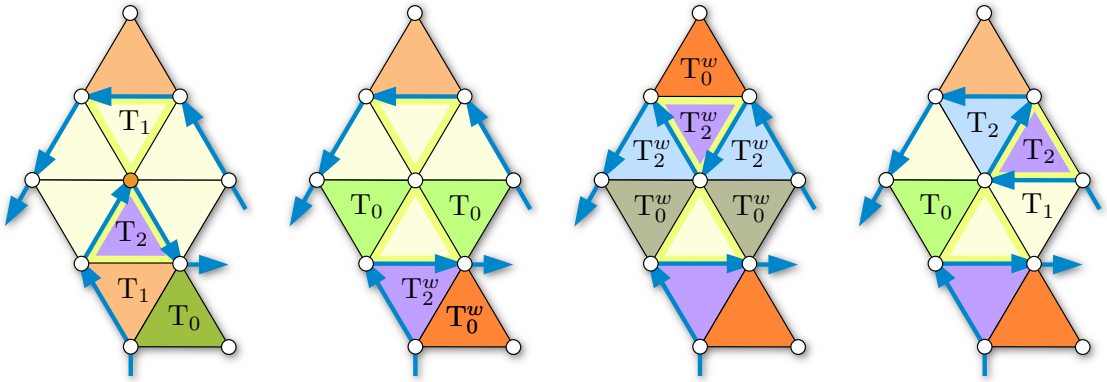
Figure 31: Code for decoding $v.L$ or $v.R$ of a triangle (left), example block of 32 triangles $\{65, 67, \ldots, 127\}$ (center), and corresponding execution of the code (right) for triangle 113. The triangle numbers for exceptions (e.g. $65, 71, \ldots$) are marked red. The 128-bit fixed-size block data along with five 32-bit key vertices encode this block using 9 bpt.

7.5 Ring-Bender

In addition to reducing storage for T_0 triangles, we may also reduce their frequency. We observe that every T_0 split triangle is connected to one or more T_2 triangles by a series of T_1 triangles that we call a **branch**. By applying the Ring-Bender algorithm, we iteratively shorten branches until the T_2 and T_0 become adjacent, and thus can be encoded as an inexpensive wart. We apply Ring-Bender after constructing the ring using Ring-Expander [22] and before delta encoding.

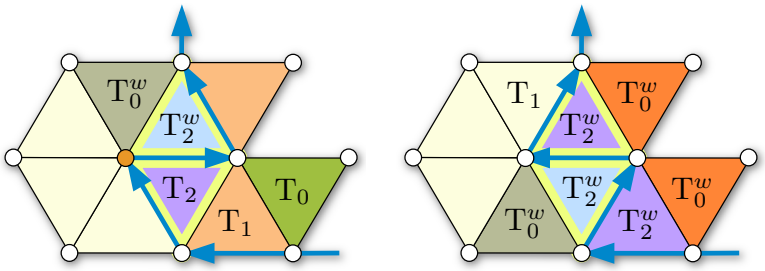
To remove the split triangles, we reroute the ring, starting from the vertex v shared by both ring edges in a T_2 triangle (highlighted in Fig. 32 and Fig. 33). In effect, this rerouting "flips" the T_2 to the other side of the ring, shortens the branch bounded by the T_2, and leaves v isolated. To bring v back into the ring, we flip one of its incident T_1 triangles. This procedure is performed iteratively until the branch is eliminated and a wart is created. Note that we allow a pair of triangles to be flipped so long as no new T_0 splits are introduced.

We show the result of a single step of Ring-Bender in Fig. 32. Notice that the T_0 triangle at the bottom is converted to a T_0^w, allowing us to encode it as a ring triangle. Ring-Bender may create as many as three T_0^w triangles for each T_0 triangle, but because Zipper encodes T_0^w as if they were T_1 triangles, using this solution does



(a) before *(b) flip 1* *(c) flip 2* *(d) bad flip 2*

Figure 32: A single step of Ring-Bender results in exchanging a pair of triangles between the two sides of the ring (we say that we "flip" them). (a) We begin at the marked vertex of a T_2 triangle, (b) flip the T_2 to make a wart, and (c) make another flip to convert the two new T_0 triangles into warts. (d) If we flip the purple T_2 in the second step, we will fail to make warts of the T_0 triangles.



(a) before *(b) after*

Figure 33: Ring-Bender zig-zag configuration corresponding to the clause on line 8 in listing 7.1. Here, the flipped triangles are an adjacent T_2/T_2 pair, rather than the non-adjacent T_2/T_1 pair in Fig. 32.

not increase the storage or execution cost.

We described in the previous paragraph the most common configuration. The special situation when two T_2 triangles are adjacent is shown in Fig. 33. Our algorithm handles such cases in listing 7.1 (line 8).

```
 1: do {
 2:  changed = false
 3:  for tri in triangles(mesh) {
 4:    if isT2(tri) {
 5:      tip = tipVertex(tri)
 6:      for neighbor in incidentTriangles(tip) {
 7:        if (isT1(neighbor) && !adjacent(neighbor, tri))
 8:        || (isT2(neighbor) && adjacent(neighbor, tri)) {
 9:          flipSides(tri)
10:          flipSides(neighbor)
11:          if anyT0(adjacentTriangles(tri)) {
12:            flipSides(tri)
13:            flipSides(neighbor)
14:          } else {
15:            changed = true
16:  } } } } }
17:} while changed
```

Listing 7.1: Ring-Bender code

7.6 Implementation details

The most important change to the implementation from LR is the computation of the $v.L$ and $v.R$ references. We explain here how we compute the references to the tip vertex for a given ring triangle t. C code for this computation and an accompanying example are presented in Fig. 31.

As in LR, we number left and right ring triangles interleaved: triangles on the left have even indices; those on the right are odd. For a given ring triangle t, we

first identify the block $i = \lfloor \frac{t}{2 \times 32} \rfloor$ and the index $j \in \{0, \ldots, 31\}$ within the block associated with t. We store the two delta bits and the single wart bit separately as three consecutive 32-bit words, such that bit j within each word is associated with triangle $31 - j$ within the block (i.e. the most significant bit corresponds to the first triangle). We then fetch and bitwise AND the delta and wart words to compute an exception mask e (recall that exceptions are assigned the 3-bit binary code 111). To determine the number of exceptions that precede t within the block, we first shift out any exceptions that follow t and then count the number of set bits remaining. Bit counting can be done in constant time using the SSE4 POPCNT assembly instruction, accessible via the `gcc __builtin_popcount()` function. (Current Intel, AMD, and nVIDIA processors have hardware support for the POPCNT assembly instruction.) The result is an index into the exception list where the most recent key vertex v is stored. We then form another mask d that has all bits set for triangles between t and the exception, i.e. d flags those deltas that require summation. This summation is again accomplished in constant time using bit counting of the low and high delta bits. The accumulated delta is then subtracted off from the key vertex. Each of these steps can be accomplished in constant time using no branches or loops. Our implementation of this process is extremely efficient: it compiles to only 33 assembly instructions.

CHAPTER VIII

RESULTS

In this chapter, we report the *storage* and *performance* results of the four techniques.

8.1 Storage

In this section, we describe the storage statistics for our four representations. We first describe the meshes used in our tests then compare the results, and finally report *storage results* for each of the methods.

8.1.1 Meshes

We use a benchmark of ten models, shown in Fig. 34 and Table 1 for our evaluation of storage. The triangle count for each mesh ranges from 69K triangles to 55.5M triangles. For each mesh, the regularity of the vertices (percentage of vertices with valence 6) ranges from 32% to 86%. Each row in Table 1 lists the storage results for each benchmark mesh for all our data structures.

8.1.2 Storage analysis and measure analysis

In this subsection, we describe the connectivity storage cost for each one of our data structures.

SOT: In SOT, we require that each vertex be matched to a unique triangle.

The storage cost is exactly 3 references per triangle.

SQuad: In SQuad, we pair some of the triangles into quads, and then we require that each vertex be matched to a unique quad or unpaired triangle. To reduce storage, it is important to try and reduce the number of unpaired triangles. In Meshlets [27], we have formulated this minimization problem as a Maximum Independent Set (MIS)

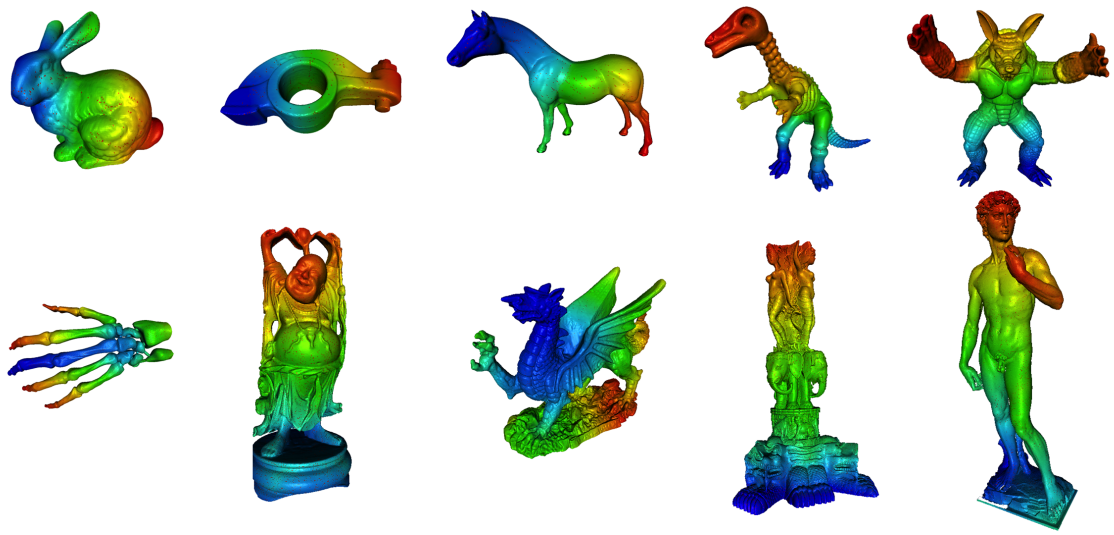


Figure 34: The ten models used in our tests. The models are (left-to-right, top-down): Bunny, Rocker arm, Horse, Dinosaur, Armadillo, Hand, Happy buddha, Welsh dragon, Thai statue, David. The color coding illustrates the ordering of triangles for the SQuad data structure. All meshes have zero genus except Hand (6), Rocker arm (1), Happy Buddha (104), and Thai statue (3). Bunny and David have borders.

problem, which is NP-complete.

In Table 2, we report how effective our construction scheme is in matching and pairing mesh primitives. The table also lists the fraction of unmatched and unpaired triangles for these meshes in their SQuad representation. Let f_0 be the fraction of unmatched triangles. Hence, the storage cost for the SQuad representation is $4m + 4nf_0$ references, which is $2 + 4f_0$ *rpt*, assuming $n = 2m$. As is evident from the table, f_0 is usually small, allowing 2.15 references per triangle or less to represent all of our meshes. We note that the mesh regularity in terms of the fraction of valence-6 vertices is correlated with the number of paired triangles.

LR & Zipper: In LR and Zipper, we are faced with two optimization problems. Given a Corner Table representation of a triangle mesh: (1) we wish to build a Hamiltonian cycle of the primal graph and (2) we wish to minimize the number of non-wart T_0 triangles defined by this cycle. We provide an approximate solution to the first challenge using a greedy linear time algorithm.

Table 1: Mesh statistics (number of vertices, number of triangles and percentage of regular vertices), and storage cost reported in references per triangle (*rpt*) or bits per triangle (*bpt*) for CT, SOT, SQuad, Standard LR, Bit-Efficient LR, and Zipper.

Mesh	m	n	% Val 6	CT (rpt)	SOT (rpt)	SQuad (rpt)	Standard LR (rpt)	Bit-Efficient LR (bpt)	Zipper (bpt)
bunny	34,834	69,451	75.1%	6	3	2.054	1.062	20.27	5.92
rocker arm	40,177	80,354	65.2%	6	3	2.054	1.055	18.37	5.62
horse	48,485	96,966	66.5%	6	3	2.046	1.055	21.59	5.65
dinosaur	56,194	112,384	57.9%	6	3	2.072	1.106	26.21	6.28
armadillo	172,974	345,944	52.6%	6	3	2.069	1.074	26.12	5.79
hand	327,323	654,666	53.4%	6	3	2.096	1.164	33.86	6.60
buddha	543,652	1,087,716	32.1%	6	3	2.150	1.583	45.26	13.37
welsh dragon	1,105,185	2,210,378	86.7%	6	3	2.027	1.020	26.12	5.21
thai statue	4,999,996	10,000,000	44.4%	6	3	2.111	1.260	34.35	7.97
david	27,812,954	55,514,795	51.6%	6	3	2.082	1.089	28.89	6.04
median			55.6%	6	3	2.070	1.082	26.16	5.98
mean			58.5%	6	3	2.076	1.147	28.10	6.84

Table 2: Mesh statistics (number of triangles and regular vertices) and SQuad representation (unmatched triangles, matched unpaired triangles, and references per triangle).

Mesh	n	Val 6	Unmatched	Matched unpaired	SQuad (rpt)
bunny	69K	75.1%	1.2%	1.5%	2.054
rocker arm	80K	65.2%	1.3%	1.3%	2.054
horse	97K	66.5%	1.1%	1.1%	2.046
dinosaur	112K	57.9%	1.8%	1.8%	2.072
armadillo	346K	52.6%	1.7%	1.7%	2.069
hand	655K	53.4%	2.4%	2.4%	2.096
buddha	1.1M	32.1%	3.8%	3.7%	2.150
welsh dragon	2.2M	86.7%	0.7%	0.7%	2.027
thai statue	10M	44.4%	2.8%	2.8%	2.111
david	55.5M	51.6%	1.9%	2.1%	2.082

Table 3: Storage statistics for the standard and Bit-Efficient LR representation.

mesh	n	%v6	LR					Bit-Efficient LR						
			m_i	$\%T_0$	$\%T_0^w$	$\%T^i$	rpt	k	m_i	$\%T_0$	$\%T_0^w$	$\%T^l$	$\%T^i$	bpt
bunny	69K	75.1	1	0.41	1.74	1.14	1.062	226	1	0.63	1.49	3.94	1.81	20.27
rocker arm	80K	65.2	0	0.39	1.73	1.06	1.055	720	0	0.40	1.79	1.71	1.10	18.37
horse	97K	66.5	1	0.39	1.48	1.05	1.055	226	0	0.61	1.53	8.61	1.72	21.59
dinosaur	112K	57.9	0	0.75	2.30	2.02	1.106	106	3	1.35	2.34	12.83	3.66	26.21
armadillo	346K	52.6	3	0.52	2.42	1.41	1.074	89	10	1.26	2.51	13.72	3.44	26.12
hand	655K	53.4	11	1.17	3.15	3.12	1.164	68	16	1.92	3.21	28.84	5.15	33.86
buddha	1.1M	32.1	180	4.26	3.57	10.95	1.583	38	273	4.78	3.57	26.12	12.22	45.26
welsh dragon	2.2M	86.7	4	0.14	0.82	0.38	1.020	100	5	0.87	1.05	18.84	2.52	26.12
thai statue	10M	44.4	241	1.85	3.12	4.96	1.260	69	298	2.39	3.12	23.73	6.42	34.35
david	55.5M	51.6	1143	0.63	3.19	1.64	1.089	108	1623	1.21	3.10	23.06	3.28	28.89

We address the second challenge in Zipper only using a sequence of local greedy improvements performed by our RingBender algorithm that modifies the ring locally.

LR: Let the mesh have a ring with m_r vertices, leaving $m_i = m - m_r$ vertices isolated. Let n_0 be the number of T_0 triangles remaining after wart skipping, and let n^i be the number of T_1^i and T_2^i triangles. In the LR representation, we store a total of $2m_r$ references in the LR table, $6n_0$ references in the VO table, $3n^i$ references in the VO^* table, and m_i references in the C table. Hence, the storage cost for the LR representation is $2m_r + 6n_0 + 3n^i + m_i$ references, which is $\mathbf{1}+f_1$ *rpt* , where f_1 is $(6n_0 + 3n^i - m_i)/(2m)$, assuming $n = 2m$ (since $2m_r + 6n_0 + 3n^i + m_i = 2m_r + 2m_i + 6n_0 + 3n^i - m_i = 2m + 6n_0 + 3n^i - m_i$ (since $m_r + m_i = m$), therefore $(2m + 6n_0 + 3n^i - m_i)/n = 1 + f_1\ rpt$).

Table 3 lists n, the number of triangles and m_i, number of isolated vertices, the percentage of valence-6 vertices and T_0, T_0^w, and T^i triangles, as well as the corresponding *rpt*. The median storage cost for LR is **1.08** *rpt* , which is about half the storage cost for SQuad. As in SQuad, the storage cost is influenced by the regularity of the mesh, and is proportional to the fraction of valence-6 vertices.

Zipper: Our Ring-Bender technique, though simple, is quite effective at converting expensive T_0 triangles to cheap warts. Ring-Bender reduces the median fraction of T_0 triangles (relative to the total number of triangles) from 0.574% to 0.249%—a reduction of 2.3x—while increasing the ratio of warts from 2.25% to 3.24%. The reduction in fraction of T_0 triangles reduces the storage cost by roughly 1 *bpt* on average.

The storage cost for Zipper can be expressed in terms of the number of ring triangles n_r, conditional exceptions n_e (i.e. not including the first key vertex in each block), T_0 triangles n_0, opposite corners n_c that cannot be inferred, total mesh triangles n, and the hash load factor f_2. Since the hash stores only T_0 corners, $n_c = 3n_0$. Thus the total cost is $\frac{32}{n}\left(5\lceil\frac{n_r}{32}\rceil + n_e + 3n_0 + \frac{1}{f_2}n_c\right) \simeq 5 + 32\frac{n_e+6n_0}{n}$ *bpt*, where

Table 4: For each mesh we indicate its triangle count n and percentage of valence-6 vertices; the percentage of delta values 0–3 and exceptions; the Zipper storage cost for fixed-size block data, conditional key vertices, T_0 vertex references, T_0 opposite references; and the total Zipper, CT, SQuad, LR and BELR storage cost (in bits per triangle) and ratio relative to Zipper.

mesh	n	%v6	delta frequency (%)					Zipper storage cost (bpt)				
			Δ_0	Δ_1	Δ_2	Δ_3	ex.	block	key	T_0 v	T_0 o	total
bunny	69K	75.1	23.8	51.8	17.7	2.3	4.5	5.022	0.426	0.231	0.238	5.92
rocker	80K	65.2	26.0	48.4	18.7	2.9	4.0	5.006	0.286	0.161	0.167	5.62
horse	97K	66.5	23.7	52.3	17.6	2.4	4.0	5.006	0.279	0.179	0.186	5.65
dinosaur	112K	57.9	29.3	43.3	18.8	3.7	4.9	5.006	0.572	0.344	0.356	6.28
armadillo	346K	52.6	30.3	41.2	20.2	4.0	4.3	5.002	0.381	0.198	0.205	5.79
hand	655K	53.4	32.5	38.0	19.6	4.4	5.5	5.000	0.754	0.416	0.430	6.60
buddha	1.1M	32.1	39.1	26.9	16.4	5.1	12.6	4.997	3.016	2.635	2.720	13.37
welsh	2.2M	86.7	21.7	54.9	18.6	1.3	3.4	5.000	0.094	0.058	0.059	5.21
thai	10M	44.4	35.4	33.5	18.7	5.3	7.1	5.000	1.264	0.839	0.866	7.97
david	55.5M	51.6	28.2	45.1	18.0	3.8	4.8	5.010	0.533	0.247	0.254	6.04
median		55.7	28.7	44.2	18.6	3.8	4.6	5.004	0.480	0.239	0.246	5.98
mean		58.5	29.0	43.5	18.4	3.5	5.5	5.005	0.760	0.531	0.548	6.84

we have used the approximations $n \simeq n_r$ and $f_2 \simeq 1$. We break down the above sum into the per-triangle storage cost for blocks (including the compulsory first key vertex in a block), conditional key vertices, and T_0 vertices and opposites.

In Table 4, we report these costs for our benchmark meshes. Similar to SQuad and LR, Zipper storage increases with mesh irregularity (i.e. fewer valence-6 vertices). As seen in this table, the median storage needed for Zipper is 5.98 *bpt*. Standard LR (with adjacency) stores on average 34.6 *bpt*, which is 5.8x more than Zipper storage.

8.2 Performance

8.2.1 Micro-benchmarks

To test the speed of our implementation of the core corner operators, we propose the following micro-benchmark tests. Each of these test various aspects of the data structures: core corner operators such as the $c.v$, $c.o$ or $c.s$ operators, accessing a local neighborhood of corners, accessing geometry information, and mesh traversal in a random access patterns.

- **Vertex operator**: This test is designed to test the speed of the incidence query, $c.v$. In this test, we sequentially iterate over all the corners of the mesh and query the vertex ID for each corner. For the Corner Table, $c.v$ is a simple table look-up, but for SOT, the $c.v$ operator requires swinging around a vertex. For SQuad, a similar swinging around a vertex is performed, but less swings have to be performed. For LR and Zipper, the vertex ID is either inferred or retrieved based on the corner ID.

- **Triangle adjacency operator**: This test is designed to test the speed of the adjacency queries, $c.o$ or $c.s$. In this test, similar to the vertex operator test, we iterate over all the corners of the mesh and query the adjacent corner ID for each corner. For SOT, LR and Zipper, we compute the opposite corner $c.o$ as the adjacent corner, whereas for SQuad, we compute the swing corner $c.s$ as the adjacent corner. The adjacency operator computes the ID of a corner in an adjacent triangle. Both the $c.s$ and $c.o$ operators compute such adjacent corner information. We choose $c.s$ for SQuad, but $c.o$ for others because for SQuad, the $c.s$ operator is stored in the look-up table (to make the $c.v$ operator faster), whereas for SOT, $c.o$ is stored in the look-up table. For LR and Zipper, $c.o$ is inferred from local triangle configurations. Therefore, we feel it is fair to compare the $c.s$ and $c.o$ operators as the adjacency operator as both give a corner in an adjacent triangle, and that one can implement one from the other using the $O(1)$ $c.n$ operator.

- **Valence**: This test is designed to test the speed of accessing a local neighborhood of triangles around a vertex. In this test, we iterate over all vertices and visit all triangles incident on each vertex, from which the valence can be derived. Note that there are alternative ways to compute the valence of all vertices, e.g. a global approach of maintaining an array of valences where we

visit each corner and update the valence of the incident vertex. But we point out that the goal of the valence micro-benchmark is not to compute the valence, but to locally *access* the triangles incident on a given vertex. This test accesses only the connectivity.

- **Vertex normal**: This test is designed to test the speed of accessing the triangles' incident on a vertex while also accessing the geometry information for the triangle's vertices. It is similar to the Valence test but unlike the Valence test, in the Normal test, the geometry table is also accessed. In this test, as in the Valence test, we iterate over all vertices and visit all triangles incident on the vertex, from which the normal at the vertex can be derived as the sum of incident triangle normals. This test reveals the impact of accessing both the connectivity and geometry information in a data structure.

- **Contour**: This test is designed to test speed of random access patterns, specifically, the speed of accessing triangles and their geometry information in a random pattern, where we walk from a triangle to one of its adjacent triangles in a non-deterministic order. To be specific, we follow the *contour* where the mesh intersects a plane, which involves a non-sequential, data-dependent traversal. Let us describe the test in detail. We first pick a starting triangle s. Then pick a corner c of the triangle. We pick a value, z defined as the average of $c.g.z$ and $c.p.g.z$ and define the plane P as the xy-plane of value z. Then we walk from s to its adjacent triangle, specifically to corner $c.n.o$. We then test the edges incident on corner $c.n.o$ and pick the edge e that intersects the plane P. We then continue the walk to the adjacent triangle, specifically the corner opposite to edge e. We then repeat the steps listed above, i.e. first identify an edge intersecting the plane P, then continue the walk to an adjacent triangle as described above. We continue the walk until we return to the original starting

triangle, or if the walk terminates at a border edge. Also, the set of starting triangles for the walk are defined as all triangles with triangle ID $i * n_S$, where i and n_S are both integers.

For all the micro-benchmark tests, we compute the total time t each test requires. We also make note of the total number of elements e we access during each test, where e is defined as the total corners for the Vertex and Adjacency test, the total number of vertices for the Valence and Normal test, and the total number of triangles visited in the Contour test. We then report the time required per operation as t/e.

8.2.2 Test Mesh

We perform the tests on the David mesh, consisting of 55.5 million triangles.

8.2.3 Test configuration

The machine we use for our tests is a 2.66 GHz Intel Core i7 MacbookPro with 8 GB of 1067 MHz DDR3 memory. Our code was compiled using gcc 4.6.1 with the -O3 and -msse4 compiler flags.

During a test, we vary the amount of total RAM available to the machine. RAM availability can be configured at boot time. Note that the operating system reserves at least 400 MB for system purposes, hence, the actual amount of memory available for our tests is less than the total RAM.

8.2.4 Performance results

We compare the performance of our data structures to the Corner Table. Our performance results are listed in Table 5(e) and shown in Fig. 35. First, we discuss the case when the mesh is small and fits in RAM. In SOT, computing the $c.v$ operator requires multiple memory accesses and computations, which increase the execution time of $c.v$ for SOT by 45× over CT, while the time required to compute $c.o$ remains the same. For SQuad, the execution time for $c.s$ increases by 5× and for $c.v$ by 10× over CT.

Table 5: The table shows time taken (in nanoseconds) for the various micro-benchmarks. The first column shows the available memory (in MB). Each row corresponds to timing results (in nanoseconds) for each micro-benchmark with the noted memory. The timing results for CT, SOT and SQuad with 512MB of memory is not reported as they are extremely slow.

(a) c.v

Memory	CT	SOT	SQuad	LR	BELR	Zipper
512				127.1	141.3	7.58
720	208.7	2143.7	885.4	74.8	33.8	8.63
1024	198.9	895.8	8.97	11.42	27.7	7.63
1440	176.7	35.6	8.24	4.21	27.5	7.56
2048	27.3	35.2	8.23	4.22	27.5	7.60
2880	0.78	33.9	8.48	4.20	27.7	7.60
4096	0.77	35.1	8.23	4.22	27.2	7.60

(b) c.o/c.s

Memory	CT	SOT	SQuad	LR	BELR	Zipper
512				294.0	536.5	35.0
720	280.6	274.3	1048.4	109.7	79.0	37.1
1024	154.4	146.9	31.9	11.29	75.4	36.0
1440	148.4	10.88	3.95	9.86	74.6	35.0
2048	26.2	0.78	3.96	9.73	73.8	35.4
2880	0.77	0.76	4.08	9.83	73.9	35.4
4096	0.77	0.78	3.96	9.80	73.8	35.4

(c) valence

Memory	CT	SOT	SQuad	LR	BELR	Zipper
512				2134	1714	230.7
720	250.9	1923	711.1	110.7	228.7	229.4
1024	194.0	1120	50.3	92.4	216.9	227.6
1440	192.3	45.9	37.6	86.4	216.9	227.1
2048	82.6	47.1	37.7	85.9	216.9	227.2
2880	45.0	45.9	37.7	86.3	216.8	227.2
4096	45.0	45.9	37.6	86.2	216.9	227.2

(d) normal

Memory	CT	SOT	SQuad	LR	BELR	Zipper
512				7,936	36,602	3441
720	994.8	3,286	1,656	6,209	33,434	1048
1024	944.3	2,830	1,100	186.3	1,850	417.9
1440	629.1	727.5	294.4	186.2	867.9	416.3
2048	310.5	705.8	294.9	186.3	867.9	416.4
2880	180.4	705.8	294.7	186.2	867.9	416.4
4096	185.1	705.7	294.6	186.1	868.2	416.4

(e) contour

Memory	CT	SOT	SQuad	LR	BELR	Zipper
512				66,652	137,553	27,460
720	704,728	131,987	91,921	45,252	87,989	9,855
1024	671,768	110,759	59,606	67.6	278.7	111.2
1440	425,756	192.4	109.8	68.1	188.4	111.2
2048	40,420	188.3	109.7	68.0	188.2	111.2
2880	70.6	188.7	109.9	68.1	188.3	111.2
4096	72.3	188.6	109.8	68.1	188.2	111.2

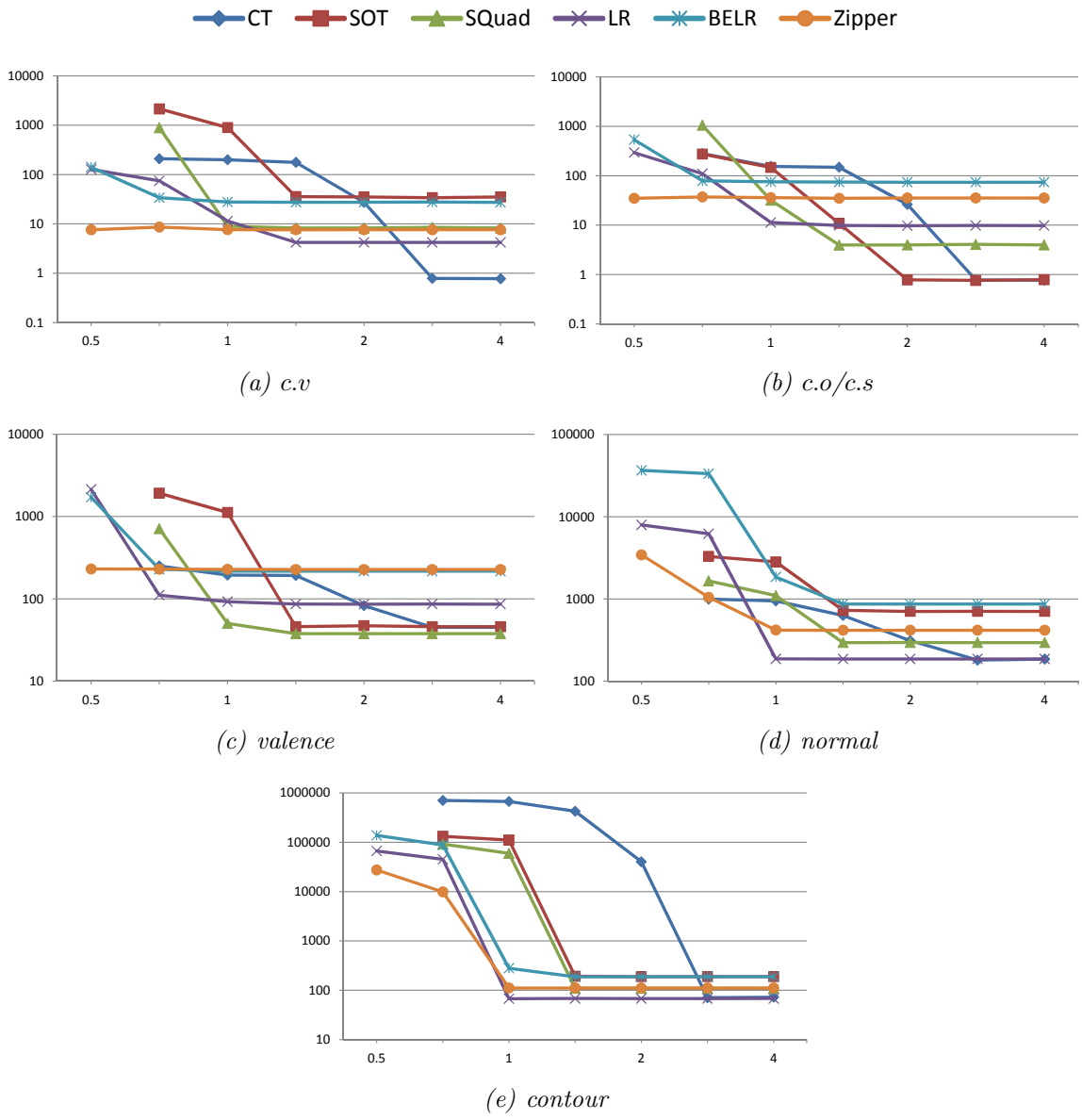


Figure 35: Per-element execution time (in nanoseconds, y-axis) as a function of available main memory (GB, x-axis) for various micro-benchmarks.

For LR, the execution time for $c.v$ increases by 5.5× and for $c.o$ by 13× over CT. For BELR, the execution time for $c.v$ increases by 35× and for $c.o$ by 96× over CT. Note that the BELR code is not highly optimized. For Zipper, the execution time for $c.v$ increases by 10× and for $c.o$ by 46× over CT. Also, the Zipper $c.v$ operator is 1.8× slower and the $c.o$ operator is 3.6× slower than in LR, while being 2–3 times faster than their BELR counterparts. A naive version of Zipper, which loops over runs of vertices to sum up deltas, runs ten times slower than the optimized version reported here.

We evaluate the performance by reducing the memory available to the testing machine in our experiments. When CT and our data structures fit in memory, CT performs better than our data structures. If CT does not fit in memory, the performance degrades due to page faults. As our data structures are more compact than CT, the operators in our data structures start performing better than their counterparts in CT. As seen in Fig. 35, compactness is beneficial when less memory is available, as page faults eventually dominate the access time. In our work, as Zipper is the most compact data structure, it provides the highest performance when limited memory is available.

The $c.v$ operator in SOT is slower than in SQuad for 3 reasons: (1) SQuad does fewer swings (since it swings quads) (2) swinging in SOT (to find the matched corner) requires both $c.o$ and $c.n$ operators, and (3) SQuad avoids modulo 3 when checking if a corner is matched (the SOT code has not been optimized to avoid such divisions). The timings for the $c.o$ operators for CT and SOT, and for the $c.v$ operator for CT are similar, since all three are costs of a simple table look-up. The cost of $c.s$ for SQuad involves converting a triangle corner to a quad corner, a table look-up, and a reverse conversion. This overhead is amortized in the higher-level tasks, for which SQuad yields comparable or, when memory is scarce, better performance than CT.

Although the sequential $c.v$ and $c.o$ table lookups in CT are faster than their

counterparts in LR when the mesh fits in main memory, this performance difference is not observed when considering higher-level tasks that require some level of random access, e.g. to access neighboring vertices. The reason being that when performing random access tests, caching is a bigger problem than the speed of the individual operators. In the random access case, Zipper and LR are generally *faster* than both CT and SQuad due to its smaller memory footprint and improved cache utilization.

8.2.5 Shortest Path test

Although these micro-benchmarks test the performance of the mesh access operators, they may not be representative of actual mesh processing applications, which normally maintain additional data structures and perform more complex computations.

As one example application, we implemented Dijkstra's (Euclidean) shortest path algorithm between mesh vertices, which requires access to geometry, the ability to mark visited vertices, and maintenance of a priority queue. We timed this application on the Welsh dragon mesh which has 2 million triangles. In our tests, we use Surface-Mesh [39] for comparison. We divide the running times for the various data structures by the running time for the Surface-Mesh data structure, therefore, Surface-Mesh uses 1 unit of time. The Corner Table uses 0.85 units of time, LR 0.83 units and Zipper 1.18 units. CT and LR are faster than Surface-Mesh by about 15%, while Zipper is slower by about 18%.

CHAPTER IX

DISCUSSION

9.1 Already existing extensions

We present here the various extensions that have been made to our data structures.

9.1.1 SOT for Tetrahedral meshes

Gurung and Rossignac [24] have extended SOT to support tetrahedral meshes. In that work, an extension of the Corner Table and of corner operators to tetrahedral meshes is presented. The Corner Table extension for tetrahedral meshes requires 8 references per tetrahedron (*rptet*): 4 for vertex IDs and 4 for opposite corner IDs. Just as in SOT, in the tetrahedral version the vertex table is eliminated and is inferred, while only the Opposite table is stored, resulting in storage of 4 *rptet*. In SOT for tetrahedral meshes, the $c.v$ operator is computed in a similar manner as in SOT: The *star* of the vertex is traversed until a matching corner is found. The star of the vertex is traversed by visiting all incident tetrahedra only once by keeping a flag per tetrahedron to indicate if it has been visited. A matching corner is defined as the first corner of a tetrahedron whose ID is less than m, where m is the number of vertices. The authors report that when computing the $c.v$ operator, on average, about 13.3 tetrahedra are visited.

9.1.2 Meshlets

Luffel, Gurung, Lindstrom and Rossignac present an extension of SQuad called Meshlets [30]. In Meshlets, a streamable SQuad is presented. In our SQuad, the triangles and quads are matched to vertices, but the ordering of the vertices is not constrained. Instead, the vertices are ordered by the application as it sees fit. In Meshlets the

information for the vertices and triangles are interleaved. In SQuad the unmatched triangles are placed at the end of the triangle list. These unmatched triangles disrupt the order of the triangles. This is fixed in Meshlets where such unmatched triangles are placed in the order of the vertices.

9.1.3 Editable SQuad

Castelli Aleardi, Devillers and Rossignac present an extension of SQuad, called Editable SQuad (ESQ) [14]. ESQ handles dynamic updates to the triangle mesh in constant time. Specifically, three kinds of dynamic updates are handled: splitting a triangle into three triangles by inserting a vertex, edge flip, and deletion of valence-3 vertices. To accomplish these updates, all possible configurations of a quad (or triangle) and its neighbors are listed, and for each of the three dynamic updates, a case by case analysis is done. In ESQ, unlike in SQuad, a quad can be matched to 2 vertices in addition to the configurations in SQuad.

9.2 Possible extensions

9.2.1 Dynamic LR

Our data structures work with static triangle meshes. As noted above, ESQ addresses three kinds of dynamic updates to SQuad. Here, we present two kinds of dynamic updates that can be handled in LR. If a non-ring edge e is flipped, then we can update the data structure locally. The two triangles sharing edge e can be of type T_0, T_1 or T_2. All 9 possible types of these two triangles can be analyzed. For each configuration, a corresponding update can be performed locally.

Another dynamic change that can be handled in LR is the edge-collapse of ring edges. The collapsed vertex is marked as deleted using 1 bit per vertex. Also, the triangles incident on the ring-edge can be deleted, while information for neighboring triangles can be updated locally.

It is unclear how we can address other kinds of dynamic updates (e.g. flipping a

ring-edge would result in the ring being broken). In such cases, it is unclear how the broken ring can be fixed. Also, a vertex split of a ring edge results in the introduction of a new vertex. It is unclear how the new vertex can be included in the ring in constant time.

9.2.2 Zipper with 64-vertex runs

Zipper uses 32-vertex runs. Alternatively, we can use 64-vertex runs, which according to our estimate should reduce storage by about 1 *bpt*. To use 64-vertex runs, we have to use the 64-bit popcount function. We performed a comparison between the 32-bit popcount and 64-bit popcount functions and noticed similar performance.

We first explain the storage cost for Zipper and the reduction in storage, and then discuss the performance of using 64-bit popcount vs 32-bit popcount.

9.2.2.1 Storage

Zipper's storage cost per run can be broken into the storage of 8 components:

(a) key vertex pointer

(b) first key vertex

(c) low delta code

(d) high delta code

(e) wart delta code

(f) conditional exceptions

(g) T_0 triangles

(h) hash table entries

In Zipper, when using runs of 32 delta codes, (a)-(e) contribute a total cost of 5 *bpt*. Let us elaborate on this cost. (a) represents the key vertex pointer, which is stored as a 32-bit pointer into a table containing exceptions (key vertices). We use 32-bit pointers because word aligned references improve performance. (b), the first

key vertex, is represented as a 32-bit reference. A combination of (c), (d) and (e) represent the 3-bit delta codes. (c), (d) and (e) are each stored as 32-bit integers to represent the 32 delta codes. Hence, the total storage cost for a single run of 32 delta codes is 5*32 bits. As each delta code corresponds to a triangle, the per triangle storage cost is 5 *bpt*. Note that the average storage cost for (f)-(h) is about 1 *bpt*, therefore the total storage cost is about 6 *bpt*.

Now, let us investigate the storage cost for switching to 64-vertex runs. In 64-vertex runs, using a similar explanation as above, (a)-(e) contribute a total cost of 4 *bpt*. Let us elaborate on this cost. The cost for (a) and (b) remain the same, while (c), (d) and (e) are each stored as 64-bit integers to represent the 64 delta codes. Hence, the total storage cost for a single run of 64 delta codes is 2*32 + 3*64 bits. As each delta code corresponds to a triangle, the per triangle storage cost is 4 *bpt*.

Note that when moving from 32-vertex runs to 64-vertex runs, the number of conditional exceptions might increase. The reason being that in Zipper with 32-vertex runs, every 32^{nd} key vertex is forced to be an exception, i.e. the first key vertex is a forced exception. When moving to 64-vertex runs, every 64^{th} key vertex is forced to be an exception. But the 32^{nd} key vertex k within a 64-vertex run, which in Zipper with 32-vertex run used to be a forced exception, might now instead become a conditional exception. We now discuss the probability that k is stored as a conditional exception. In Zipper with 32-vertex runs, our results indicate that the probability a certain delta code is an exception is about 5%. Therefore, we can say the expected number of times we need to store k as a conditional exception is about 5%, meaning the expected number of bits resulting from possibly storing k is about 5% of 32 bits which is 1.6 bits. Since a 64-vertex run represents 64 triangles, therefore storing k costs 0.025 *bpt* (i.e. 1.6/64 *bpt*). Therefore, the average storage cost for (f)-(h) now increases to about 1.025 *bpt*. Therefore, the total storage cost is about 5.025 *bpt*.

Hence, when moving from 32-vertex runs to 64-vertex runs, there is a saving of

Table 6: Millions of triangles per 100 MB when no geometry information is stored.

Name	Connectivity (rpt)	Bytes per triangle	Millions of Triangles (per 100 MB)
CT	6.5	26	3.8
SOT	3	12	8.3
SQuad	2.08	8.32	12.0
LR	1.08	4.32	23.1
Zipper	0.19	0.76	131.6

Table 7: Millions of triangles per 100 MB, when geometry is stored as 16-bit coordinates.

Name	16-bit Geometry (rpt)	Connectivity (rpt)	Total (rpt)	Bytes per triangle	Millions of Triangles (per 100 MB)
CT	0.75	6.5	7.25	29	3.4
SOT	0.75	3	3.75	15	6.7
SQuad	0.75	2.08	2.83	11.32	8.8
LR	0.75	1.08	1.83	7.32	13.7
Zipper	0.75	0.19	0.94	3.76	26.6

0.975 *bpt*, which is about 1 *bpt*.

9.2.2.2 Speed

When using 64-vertex runs, we need to use the 64-bit popcount function instead of the 32-bit one to perform constant time summation of the deltas. To evaluate the performance of the operators for Zipper with 64-vertex runs, we note that the primary difference between using 32-vertex runs and 64-vertex runs is the use of the 32-bit popcount vs. the 64-bit popcount functions. Therefore, we experimentally evaluated the speed of the two functions. Our tests indicate that the 64-bit popcount function is only 5% slower than the 32-bit version.

9.3 Storage

9.3.1 Number of triangles stored per 100 MB of memory

To give the reader an intuition on the number of triangles represented by our data structures, we discuss the number of triangles that can be represented by 100 MB of

Table 8: Millions of triangles per 100 MB, when geometry is stored as 32-bit coordinates.

Name	32-bit Geometry (rpt)	Connectivity (rpt)	Total (rpt)	Bytes per triangle	Millions of Triangles (per 100 MB)
CT	1.5	6.5	8	32	3.1
SOT	1.5	3	4.5	18	5.6
SQuad	1.5	2.08	3.58	14.32	7.0
LR	1.5	1.08	2.58	10.32	9.7
Zipper	1.5	0.19	1.69	6.76	14.8

memory. Table 6 shows the number of triangles that can be represented, if we do not store the geometry but only store the connectivity information for triangles. Table 7 shows the number of triangles that can be represented using our data structures including geometry information represented as three 16-bit quantized integers per vertex. Likewise, Table 8 shows expected results for storing our data structures with geometry information represented as three 32-bit floating point numbers per vertex. In Table 6, if we do not store the geometry, then 3.8 million triangles can be represented with the Corner Table (CT), whereas with Zipper, 131.6 million triangles can be represented, which is nearly 35 times more triangles than CT. In Table 7, for 16-bit geometry, 3.4 million triangles can be represented with CT whereas with Zipper, 26.6 million triangles can be represented, which is nearly 8 times more triangles than CT. In Table 8, for 32-bit geometry, 3.1 million triangles can be represented with CT whereas with Zipper, 14.8 million triangles can be represented, which is nearly 5 times more triangles than CT.

9.3.2 Possible worst case storage cost

To illustrate possible worst case storage cost for meshes with borders, we provide two examples. The first example explores possible worst case storage cost for SQuad while the second example illustrates possible worst case storage for LR and Zipper. The two examples are the worst case storage cost we have been able to find, but are not

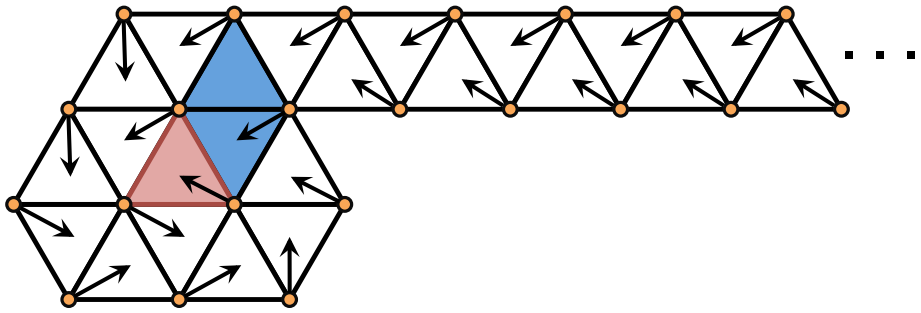


Figure 36: The example mesh contains an internal triangle (in red). The triangle strip in the right side extends infinitely (shown with dots). During SQuad construction, the blue triangles are paired as quads, while all other triangles remain unpaired.

necessarily provably worst. Therefore, we would like to point out that determining the worst case storage cost for our data structures is still an open problem. Also, our analysis of worst case storage is not intrinsic to our data structures but rather to our construction methods: the construction method listed in Fig. 14 for SQuad, and Ring-Expander for LR and Zipper.

9.3.2.1 SOT and SQuad

First, we note that during the construction of SQuad, we require a seed triangle which has 3 non-border vertices. Therefore, meshes for which all vertices are border vertices cannot be represented using SQuad. Also, note that in SOT, the number of border elements does not affect the storage cost, therefore SOT always requires 3 *rpt*.

We now discuss the example shown in Fig. 36, which is for SQuad. We first discuss the portion of the mesh that is shown in the figure, which contains 22 vertices and 23 triangles. Of the 22 vertices, 19 are border vertices while 3 are non-border vertices. When constructing SQuad using the red triangle as the seed triangle, there is one matching where a vertex is matched to a quad (shown in blue), while for all other matchings, each vertex is matched to a single triangle. As SQuad stores 4 references per matched quad or matched triangle, the total storage in Fig. 36 is 88 references. As there are 23 triangles, SQuad stores 3.82 *rpt*.

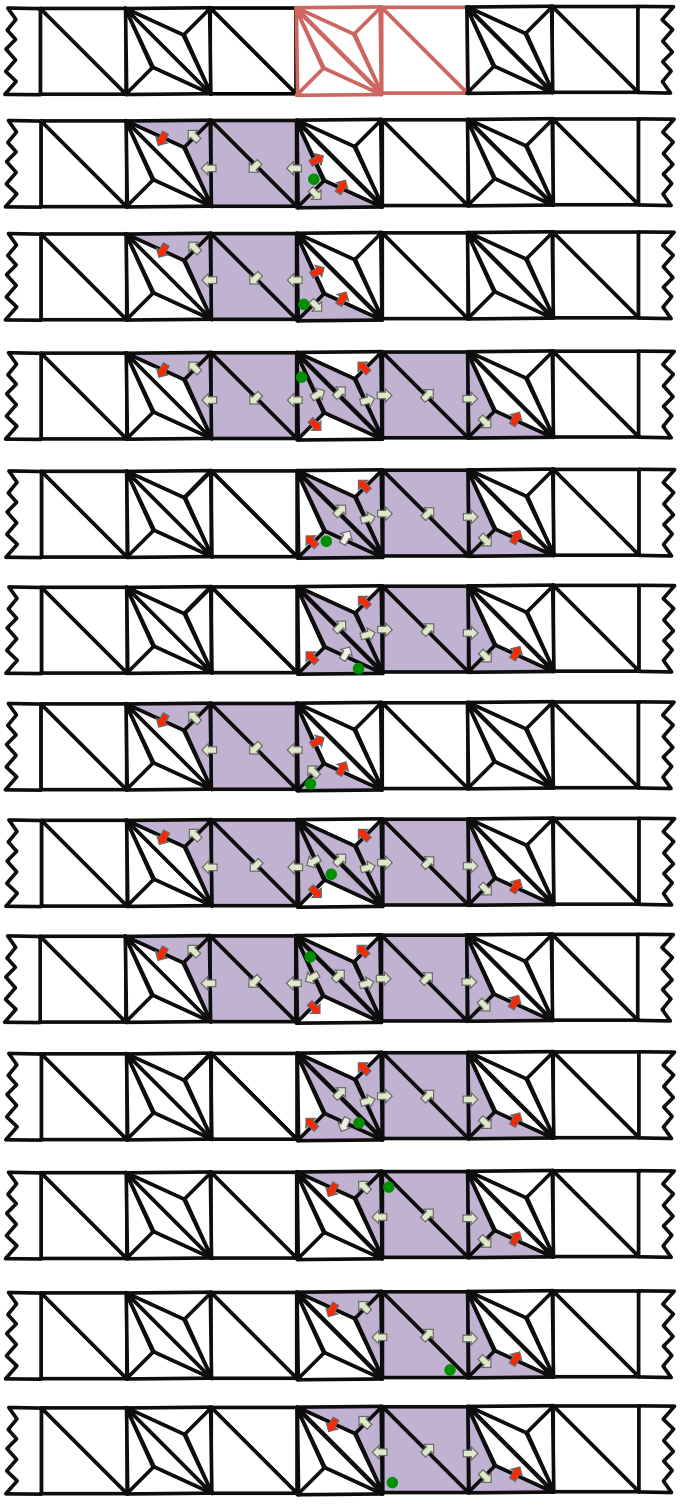


Figure 37: A repeating pattern of the triangles (shown in red, top row). The second through last rows show the visited triangles (in purple) if starting from various corners (shown in green) during construction of LR/Zipper. White arrows show valid traversal, while red arrows show triangles that are not visited.

To generalize this example, we can extend the triangle strip (on the right side of Fig. 36) to make the strip infinitely long. Note that in this example, if there are m vertices, then there are $m+1$ triangles. The storage requirement is $4m$ references as each of the m quads (or triangles) requires 4 references each. Since there are $m+1$ triangles, the storage requirement per triangle is $(4m)/(m+1)$ *rpt* which asymptotically approaches 4 *rpt*, which is about twice of what SQuad normally requires.

9.3.2.2 LR and Zipper

Let us examine the example shown in Fig. 37 to explore possible worst case storage for LR and Zipper. Note that in the construction for LR and Zipper using Ring-Expander, we start with a seed triangle and perform a depth first traversal, going to the right triangle whenever possible, going left when it not possible to go right, and backtracking whenever both are not possible.

During the construction for LR and Zipper, in the example shown in Fig. 37, if any of the corners (in green) are the starting corners when constructing the ring, we notice that the traversal visits only a small subset of the triangles. In this infinitely long mesh (extending to the left and right), the ring includes a small number of vertices (belonging to the purple triangles), and all other vertices are isolated vertices. As all triangles incident on isolated vertices are T_0 triangles, each T_0 triangle is stored using 6 *rpt*. Additionally, the $v.c$ operator is cached for each isolated vertex resulting in an additional storage cost of 0.5 *rpt*. In this arbitrarily large mesh, we can ignore the small constant storage cost for the ring vertices and its incident ring triangles. Therefore, almost all vertices are isolated vertices and almost all triangles are represented as T_0 triangles. Hence, when using Ring-Expander to construct LR and Zipper, the storage cost asymptotically approaches 6.5 *rpt*. Note that this worst case storage cost is a result of using our construction method, i.e. Ring-Expander. In the example shown in Fig. 37, we could use a different construction method which

includes all vertices in the ring for which the storage cost would be much smaller.

9.3.3 Further reduction when adjacency is not required

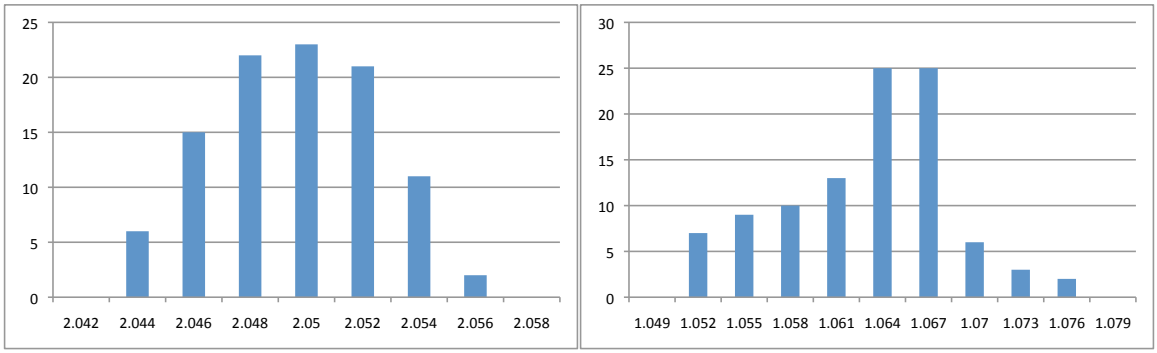
Our data structures encode both the incidence and adjacency connectivity information. Certain applications may require only the incidence information. Here, we examine how LR and Zipper may be altered to further reduce storage cost in such situations.

LR: In LR, the adjacency information for most of the triangles is inferred. If adjacency information is not needed, we may discard the adjacency information stored for T_0 triangles and for T^i triangles. For T^i triangles, instead of storing pointers to the special VO^* table, we can store the vertex ID directly. When the adjacency information is not stored, the storage cost on average will decrease by 0.06 *rpt* leading to an average *rpt* of 1.02 *rpt*, instead of the 1.08 *rpt* required for LR.

Zipper: In Zipper, as in LR, the adjacency information for most triangles is inferred, except for triangles which have T_0 triangles as adjacent triangles. For such triangles where we cannot infer the adjacency information, the adjacency information is stored in a hash table. When the $c.o$ corner is in a T_0 triangle, we store the corner in a hash table. For applications that require only the incidence information, the stored adjacency information in the hash table can be discarded. In such cases, the storage cost in Zipper reduces by about 0.25 *bpt* to a total of about 5.73 *bpt*.

9.3.4 Effect of seed triangle on storage

During the construction of SQuad, LR and Zipper, we have to choose a seed triangle. We did an experiment with the Bunny mesh where we started from a hundred random seed triangles, and listed the resulting storage for SQuad, LR and Zipper. We noticed that the resulting storage does not vary a lot. E.g. for SQuad, it ranges from 2.044 to 2.056 *rpt*, and for LR, from 1.052 to 1.076 *rpt* (see Fig. 38). During the construction of our data structures, we tried a few random starting seed triangles and stored the



(a) SQuad *(b) LR*

Figure 38: Frequency distribution (x-axis: *rpt*, y-axis: frequency) of resulting *rpt* for 100 random seed triangles for the Bunny mesh.

result with the minimum storage cost. We observed that after about 20 random seed triangles, the improvement in storage results diminish in subsequent computations starting from other random seed triangles. E.g. for SQuad, the first 20 random seed triangles give a 0.44% improvement while the last 80 random seed triangles give a 0.08% improvement, and for LR, 1.5% and 0.3%.

9.4 Impact of connectivity on storage

9.4.1 Regularity

Our storage results indicate that SQuad, LR and Zipper perform well for meshes that are highly regular, e.g. in the Welsh dragon model, 87% of the vertices are regular (valence-6) and our methods perform well with this mesh. We notice in Fig. 40 that regularity and storage results are correlated. Let us analyze the storage results for SQuad, LR and Zipper if we are given an infinitely large triangle mesh with all vertices of valence 6. For such a mesh, for both SQuad and LR/Zipper construction, the traversal spirals outwards, and has no bifurcations. For SQuad (see Fig. 41), only triangles A and F remain unpaired, but other than these two triangles all other triangles are paired. Therefore, the storage cost for SQuad will be $2 + \epsilon$ *rpt*, with ϵ being an arbitrarily small number. Likewise for LR (see Fig. 42), since there are

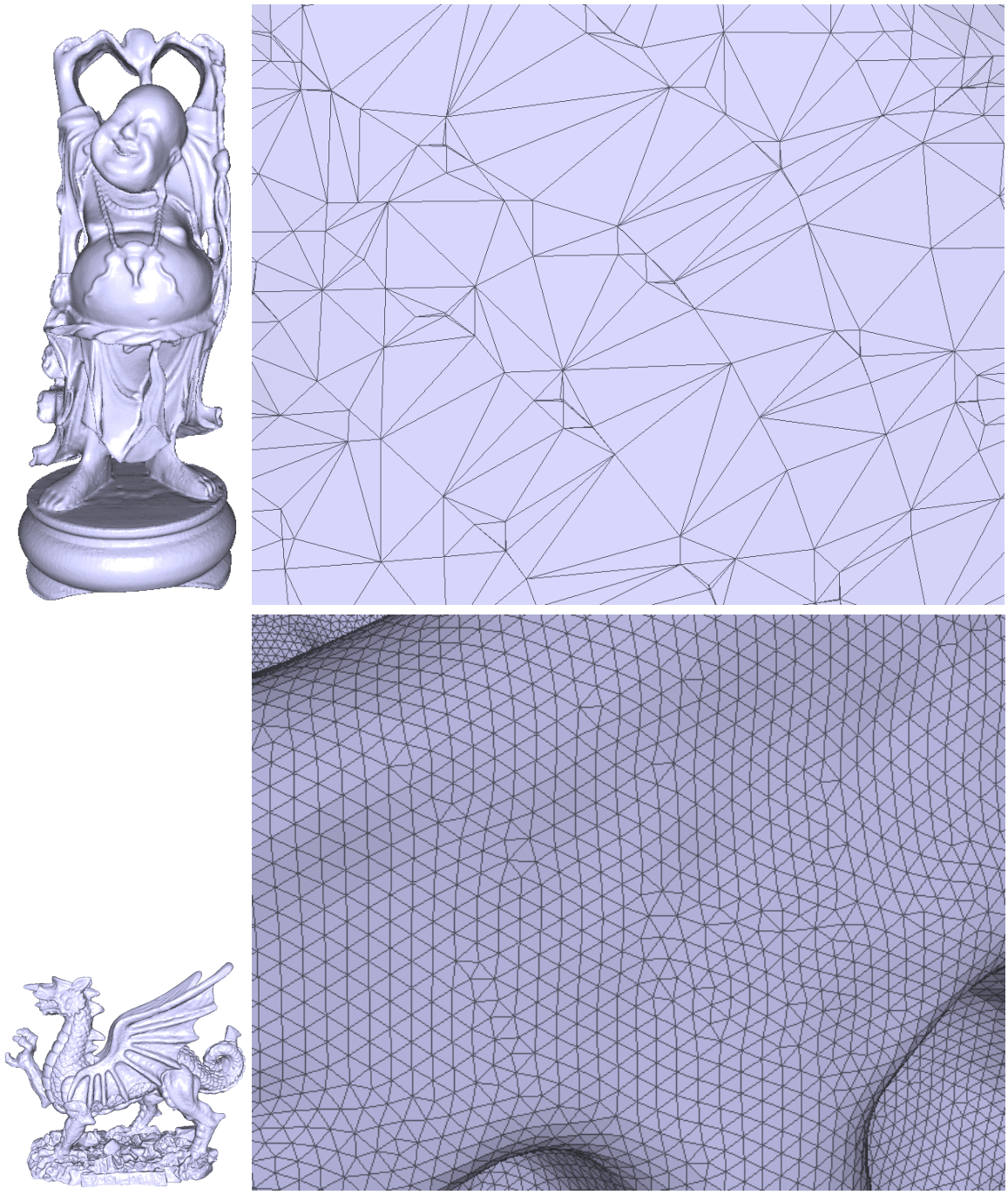
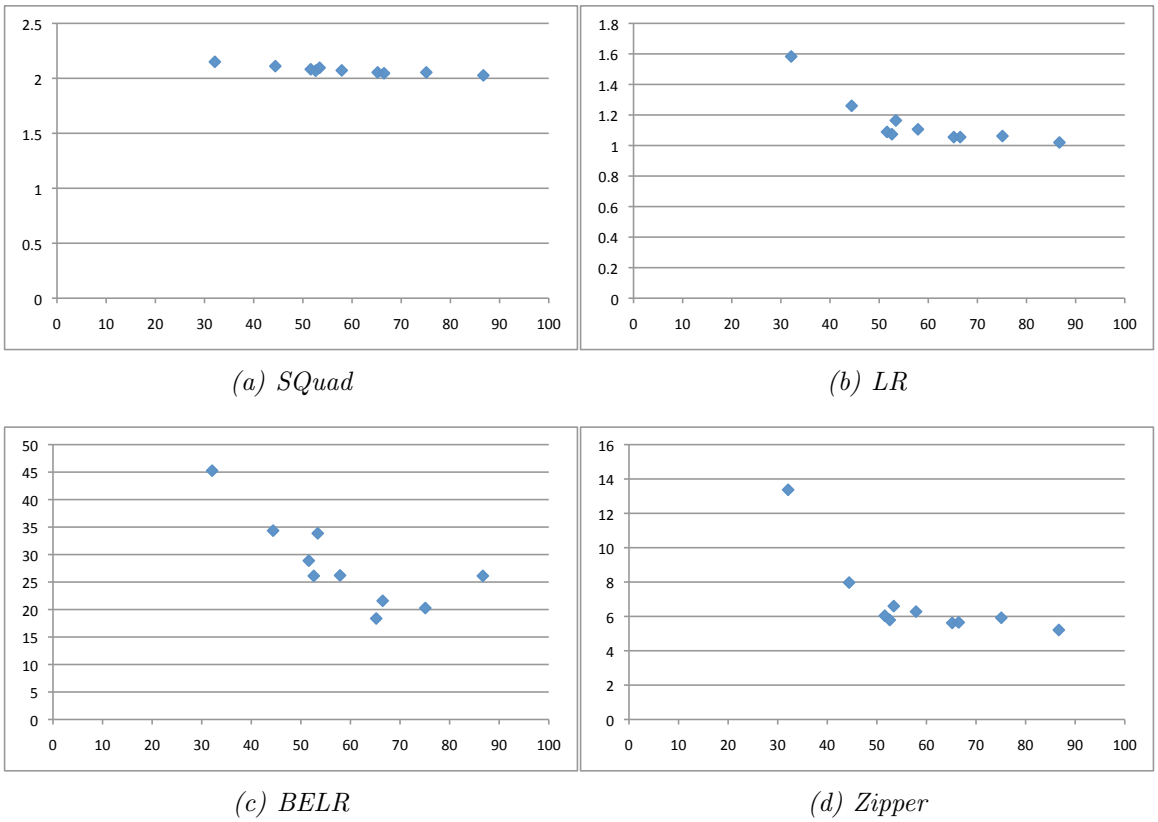


Figure 39: Closeup view of the Happy Buddha and Welsh Dragon model. The Happy Buddha has a lot of irregular vertices (only 32% regular vertices) whereas the Welsh Dragon model has a lot of regular vertices (87%).



(a) SQuad *(b) LR*

(c) BELR *(d) Zipper*

Figure 40: Plot of storage results for our 10 benchmark models (x-axis: percentage of valence-6 vertices; y-axis: *rpt* for SQuad and LR, *bpt* for BELR and Zipper). Notice that regularity (percentage of valence-6 vertices) is correlated to storage results.

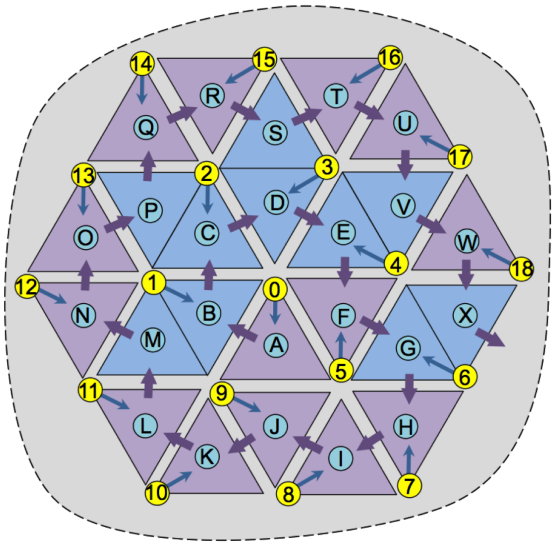


Figure 41: SQuad traversal for regular meshes.

no bifurcations during construction and only four T_2 triangles, therefore the storage cost will be $1 + \epsilon$ *rpt*. We now discuss the storage cost for Zipper. For valence-6 ring vertices with no bifurcations, we know from Fig. 27 that the deltas can only be 0, 1, 2 or 3. Hence, there are no conditional exceptions in Fig. 42 as all deltas are either 0, 1, 2 or 3. Also, there are no T_0 triangles. Therefore, the storage cost is $5 + \epsilon$ *bpt*.

If a mesh is regular, then it is likely that the storage cost will be close to the optimal result (2 *rpt* for SQuad, 1 *rpt* for LR and 5 *bpt* for Zipper). As noted earlier, an example of a highly regular mesh is the Welsh dragon model. In the Welsh dragon model, 86.7% of the vertices are regular. The storage cost is 2.027 rpt, 1.02 rpt and 5.2 bpt for SQuad, LR and Zipper respectively.

Also, highly irregular meshes can be remeshed using methods such as Swing-Wrapper [1] to produce highly regular meshes. In certain remeshing techniques, the original shape of the mesh can be preserved while improving regularity of meshes. Thus, remeshing may to lead to a decreased *rpt*.

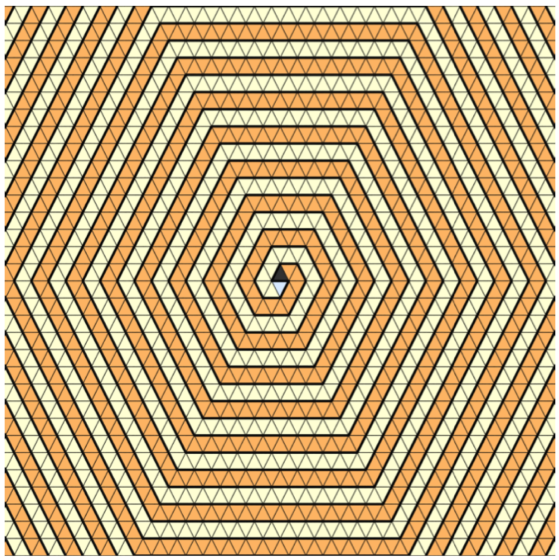


Figure 42: LR traversal for regular meshes.

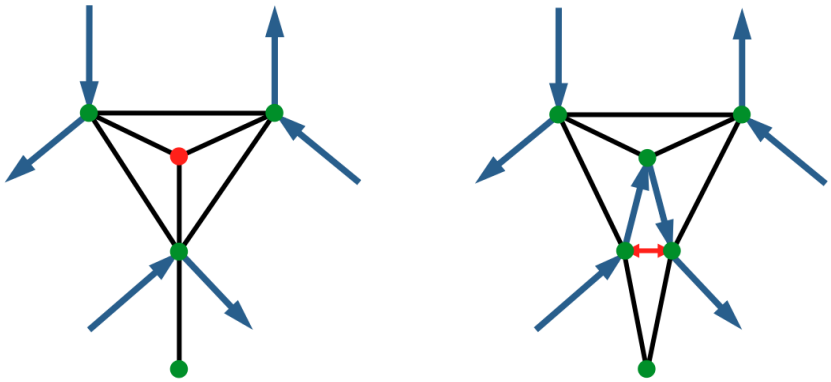
9.4.2 Hamiltonian Cycle

In the construction for LR and Zipper, a ring is constructed. In our experiments, on average 99.995% of the vertices are part of the ring, and only about 0.005% of the vertices are isolated vertices and hence are not part of the ring. In our application, if we want all vertices to be part of the ring, then we can include the isolated vertices in the ring by replicating the vertices closest to the isolated vertices (see Fig. 43). In addition to replicating the vertices, degenerate triangles are introduced.

9.5 Connectivity impact on operator timing

In SOT and SQuad, the vertex operator has a time complexity of $O(valence)$, therefore if the vertex has a high valence, the vertex operator will run slower than on a vertex with low valence. On average, the vertex operator for SOT and SQuad runs in constant time, as the average valence of a watertight manifold mesh with low genus is a constant (about 6).

In LR, as shown in Section 6.3, warts are marked using a bit. Due to this special bit, in the vertex and opposite operator for LR, there is an overhead for identifying



(a) before *(b) after*

Figure 43: Before: The red vertex is an isolated vertex, while the green vertices are part of the ring. After: The green vertex is replicated. For demonstration only, we show a split vertex separated by the red-edge. The triangles incident on the red edge are zero area triangles. The isolated vertex is now included in the ring.

wart and non-wart triangles. To improve the performance of the vertex and opposite operator, we can eliminate warts. In such cases, we do not have the overhead of the check for wart triangles. Warts can be eliminated by an edge flip of the edge between adjacent T_0 and T_2 triangles, making the T_0/ T_2 combination a T_1/ T_1 combination.

9.6 Comparison

In this section, we provide a general overview of our data structures and compare them against each other. Table 9 provides a summary of the comparisons.

Storage cost: The storage cost for SOT, SQuad, LR and Zipper are 3, ∼2, ∼1 and ∼0.19 *rpt* respectively. So the incremental improvements from one solution to the next are 33%, 50% and 81% respectively, while the improvements over ECT (6.5 *rpt*) are 2.2x, 3.1x, 6.0x and 34.7x respectively.

Implementation difficulty: The implementation of the vertex and opposite/swing operators increase in complexity when going from SOT to SQuad, to LR, and to Zipper. We briefly discuss it below.

- *vertex operator*: Among our data structures, SOT is the simplest to implement

Table 9: Comparison between our representations

Name	SOT	SQuad	LR	Zipper
rpt	3	∼2	∼1	∼0.19
Implementation difficulty: vertex operator	easiest	medium	difficult	most difficult
Implementation difficulty: opposite/swing operator	easiest	medium	difficult	difficult
Running time: vertex operator	$O(valence)$	$O(valence)$	constant	constant
Running time: opposite/swing operator	constant	constant	constant	constant
Construction: Space and running time	linear	linear	linear	linear
Order constraint: Triangles	imposed	imposed	imposed	imposed
Order constraint: Vertices	respected	respected	imposed	imposed
Extension: Streaming	no extension yet	yes	no extension yet	no extension yet
Extension: Tetrahedral meshes	yes	no extension yet	no extension yet	no extension yet
Extension: Dynamic connectivity change	constant time	constant time	for simple instances	for simple instances
Millions of triangles per 100 MB: No geometry	8.3	12.0	23.1	131.6
Millions of triangles per 100 MB: 16-bit geometry	6.7	8.8	13.7	26.6
Millions of triangles per 100 MB: 32-bit geometry	5.6	7.0	9.7	14.8
Distinguishing strength	Simple to implement	Streamable	Good compromise between space and performance	Most compact

because in SOT, the vertex operator implementation only involves swinging around a vertex. In the vertex operator for SQuad, we have to account for quads while swinging around a vertex, which makes the implementation of the vertex operator more difficult than for SOT. In LR, a case-by-case analysis of the corner and the type of triangle has to be performed to determine the vertex ID. In Zipper, in addition to the implementation in LR, the L and R references have to be evaluated.

- *opposite/swing operator*: For SOT, the opposite operator is a straightforward table look-up. For SQuad, the swing operator involves testing the corner ID to distinguish between corners for which the opposite lies in the same quad or outside the quad. For LR, local configurations of neighboring triangles have to be evaluated in order to determine the opposite corner. In Zipper, in addition to the checks made in LR, the L and R references have to be evaluated from the delta runs. Additionally, in Zipper we may need to do hashing instead of table lookups. As in the case for the vertex operator, the implementation for each of the data structures increases in complexity from SOT, to SQuad, to LR, and to Zipper.

Running time: The vertex and opposite/swing operators have an average constant time complexity for all our data structures. Note that the vertex operators for SOT and SQuad have a $O(valence)$ time complexity. We briefly discuss the running time for vertex and opposite/swing operators:

- *vertex operator*: For SOT and SQuad, the running time for the vertex operator is $O(valence)$ because when computing the vertex operator, we need to swing and visit corners incident on the vertex until we find a corner that is the first listed for the quad or triangle and has a sufficiently low ID. For LR and Zipper, the vertex operator runs in constant time because during its computation, we

analyze a fixed number of cases.

- *opposite/swing operator*: For all our data structures, the opposite operator has constant time complexity. For SOT, it is a straightforward table look-up. For SQuad, it is a table look-up combined with handling opposite corners within the same quad. For LR and Zipper, a fixed number of configurations are analyzed when determining the opposite corner information.

Construction: Here, we discuss the space requirement and running time for the construction of our data structures. During construction, we can construct the Corner Table from an indexed mesh in linear time (see Section 2.3.1). The space requirement for all our data structures is linear in the number of triangles. The running time for construction is also linear in the number of triangles. During construction, we perform a traversal and then reorder the triangles and/or vertices. The traversal is a linear-time depth first traversal of the triangles.

Order constraint: Our data structures exploit the ordering of triangles and/or vertices to infer connectivity information. All four of our data structures perform a reordering of the triangles, while the reordering of the vertices is performed by LR and Zipper only.

- *Order constraint for triangles*: In SOT, the ordering is based on the vertex-triangle matching while in SQuad, the reordering is based on the vertex-quad matching. In LR and Zipper, the triangle order is imposed by the order of the vertices along the ring.

- *Order constraint for vertices*: SOT and SQuad do not perform a reordering of the vertices, but preserve the original order. In LR and Zipper, the order of the vertices is imposed by the order of their apperance along the ring.

Distinguishing strengths: Finally, we would like to point out the defining strengths for each of our data structures. SOT is simple to implement, SQuad is

streamable, LR is a good compromise between space and performance, and Zipper is the most compact among our data structures.

CHAPTER X

APPLICATIONS TO GEOTECHNICAL ENGINEERING

10.1 Introduction

In geotechnical engineering, understanding the engineering properties of particulate materials is important to determine the strength and stability of soil, which in turn is important in the design, construction and performance assessment of buildings, roads and bridges. For example, experiments involving different shearing conditions are typically performed on sand samples to understand the change in sand structure. Traditionally, 2D image slices of the samples were studied to interpret the sand structure. Preferably, sand samples can be analyzed using their 3D voxel representation.

We discuss below how these 3D voxel representations can be segmented into sand particles, and how the bounding surface of each particle can be extracted and represented by a triangle mesh. Studying configurations that involve a non-trivial number of particles with sufficient accuracy may involve processing a model with 200 M triangles. Hence, the solutions presented in this thesis may provide benefits because they reduce the storage significantly.

10.2 Acquisition of 3D voxel representation

In our experimental study, we use the 3D voxel representations of sand particles acquired by Yang [45]. We briefly summarize the process used to acquire the data: First, a specific stress is imposed on the sand, then the structure of the sand specimen is fixed by impregnating the specimen with epoxy resin. The 3D voxel representation of the specimen can be acquired using non-destructive methods such as MRI or X-Ray CT scans, or by using destructive methods such as serial sectioning and optical

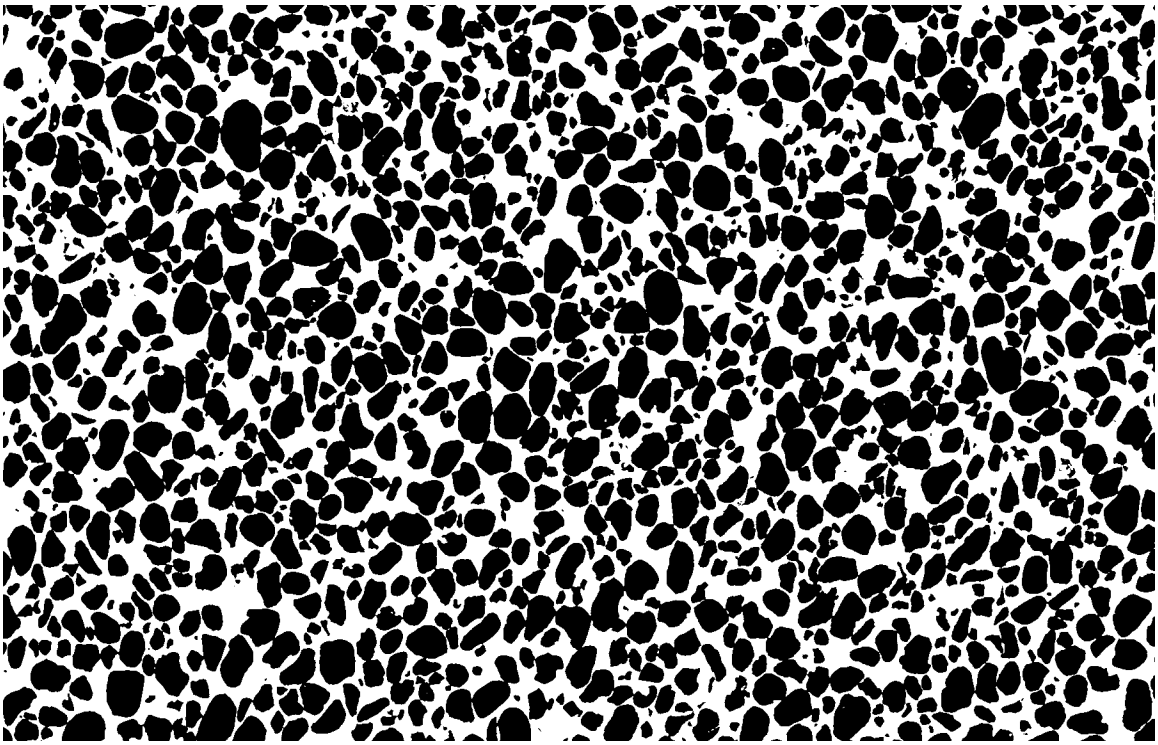


Figure 44: Example of a 2D slice of the sand specimen. The particles are presented as black pixels while the pore space is represented as white pixels. Used with permission [45].

microscopy. In Yang's work, serial sectioning is used. In serial sectioning, a thin slice of the top layer of the material is removed after which a 2D image (see Fig. 44) of the top layer is acquired. This process is repeated to generate a stack of 2D images that approximate the 3D voxel representation of the union of the sand particles (see Fig. 45 and Fig. 46).

To differentiate the sand and pore space, the 3D voxel representation is thresholded.

In Yang's work, the sample's space is divided into 1800×1100×600 voxels (∼1.2 billion voxels) and each voxel is a cube of about 8 microns in length. The model represents approximately 20,000 sand particles.

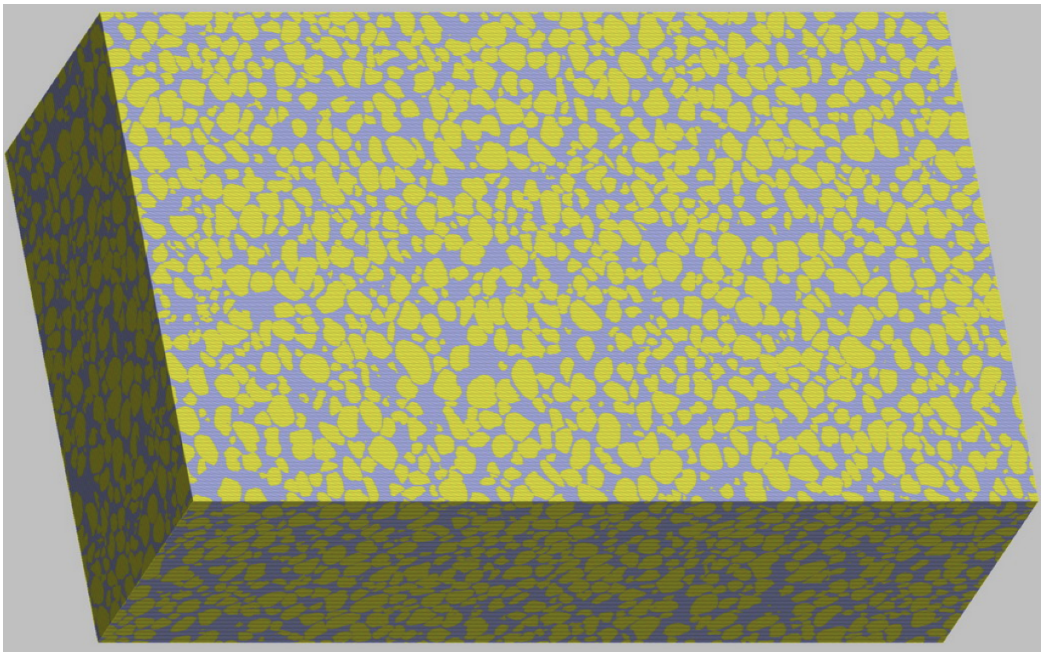


Figure 45: A parallel projection view of the model of about 20,000 sand particles (shown in yellow). The pore space is shown in gray. Used with permission [45].

10.3 Segmentation of sand particles

In the 3D voxel representation, a group of multiple sand particles is often merged into a single connected solid component.

To identify the individual sand particles, we used the following sequence:

1. a morphological shrinking [37] of the sand voxels by a given distance d,

2. the identification of the connected components, each being associated with a different sand particle and assigned a different color (for visualization)

3. a morphological growth of each sand particle by d, to undo the loss of volume produced by the initial shrinking, during which each voxel invaded during the growth is assigned to the first sand particle that claims it.

At the end of this segmentation, each voxel is either associated with a unique sand particle or is declared to be part of the pore space.

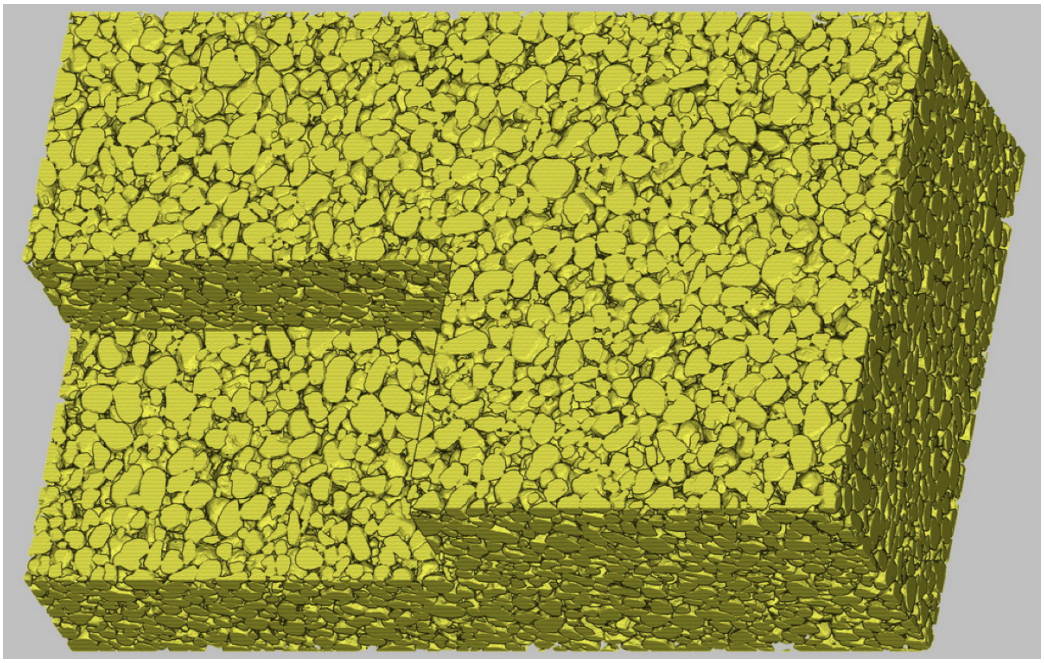


Figure 46: 3D cutout of sand particles. Used with permission [45].

10.4 Iso-surface computation

The iso-surface for each individual particle can be extracted through a variety of iso-surface extraction techniques [16]. We used an implementation of Marching Cubes [29]. The process produces a triangle mesh representing an approximation of the surface that bounds the particle. A typical sand particle can be represented by a triangle mesh consisting of about 10,000 triangles. Examples of individual particles are shown in Fig. 48 and Fig. 49.

Given that there are approximately 20,000 sand particles in the specimen, representing all the sand particles requires about 200 million triangles. Although 11-bits per coordinate would suffice to represent the exact position of the vertices of the iso-surface, we use 16 bits to represent the vertex coordinates.

To represent this triangle mesh, the Extended Corner Table requires approximately 5.8 GB of memory. SOT can represent the same triangle mesh with a storage cost of 3 GB while SQuad requires 2.2 GB of memory. The storage cost for LR is 1.5 GB while Zipper requires only 750 MB of memory. If the user requires more detailed

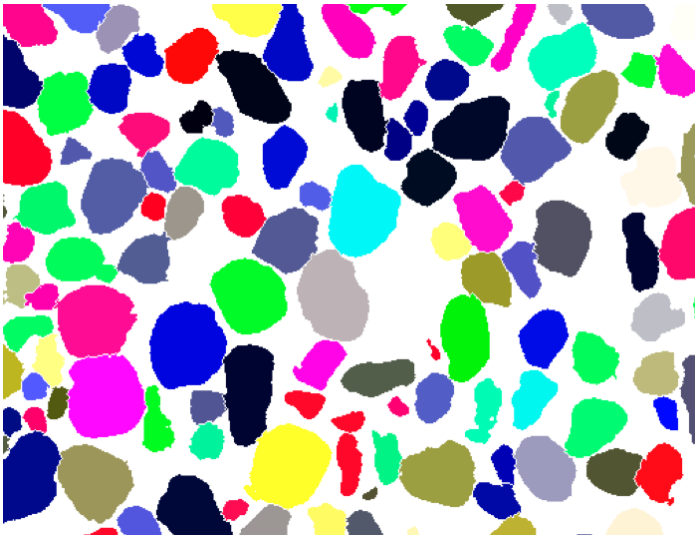


Figure 47: A 2D slice of segmented particles. Each particle is displayed using a random color to visually distinguish different particles.

models or larger samples, it results in a mesh with more triangles. In such scenarios, depending on the needs of the user and availability of RAM, the Extended Corner Table or any of our data structures can be used to represent these meshes in memory.

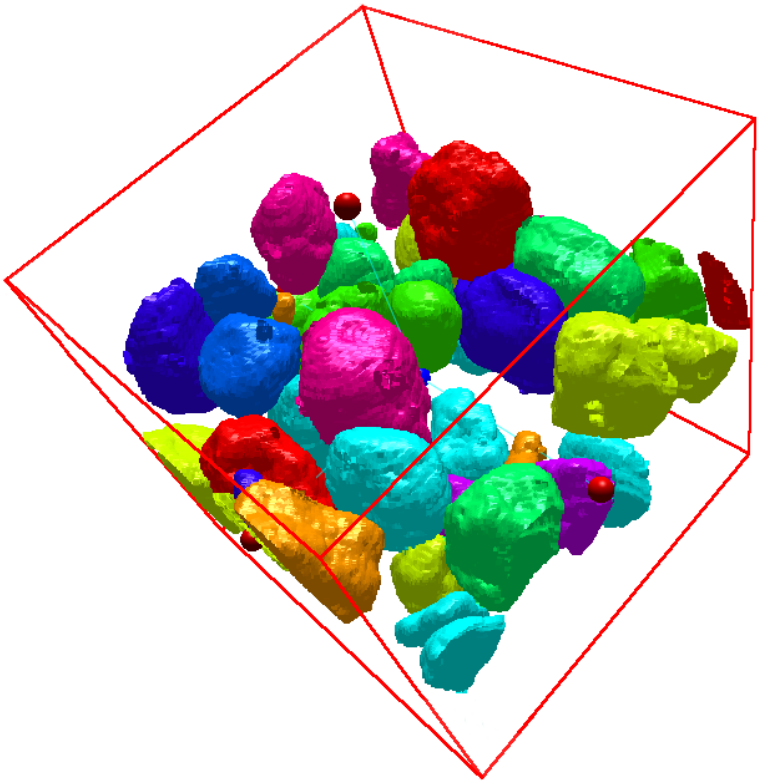


Figure 48: Sand particles where each particle is displayed using a random color to visually distinguish different particles.

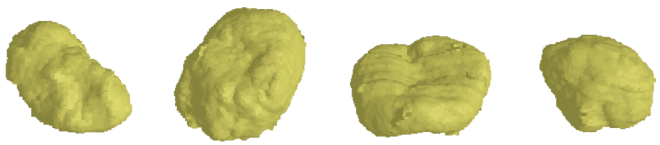


Figure 49: Four different sand particles

CHAPTER XI

CONCLUSION

In this dissertation, we have presented four new data structures for representing the connectivity of manifold triangle meshes with fixed connectivity.

Our data structures can be constructed in linear space and time from the popular face indexed format and its variations, and support all standard random-access and mesh traversal operators proposed in the Corner Table [35] in constant time (or in average constant time, in case of SOT and SQuad). They reduce the storage cost significantly over previously proposed data structures that support direct access and mesh traversal at constant time. For example, our most compact data structure, Zipper, uses only an equivalent of a fifth of a 32-bit reference per triangle (*rpt*) to store enough information to access (in constant time) not only the three vertices of each triangle, but also its three neighboring triangles, as well as one incident triangle for each vertex. Although the significant storage cost reductions achieved by our new representation schemes come at a price of a performance overhead, the corner operators they support have average constant time complexity and fast execution time. Our representations have been invented by exploiting several storage reduction techniques presented in this thesis:

- SOT (Sorted O Table) uses 3 *rpt*. The storage result is a 54% improvement over the previously proposed Extended Corner Table (ECT [35], which uses 6.5 *rpt*). SOT, which is discussed in Chapter 4, utilizes the principles of matching and reordering to reduce storage. In SOT, about half of the triangles are each matched to a different bounding vertex. Then the matched triangles are reordered to follow the order in which their matching vertices were ordered in

the input.

- SQuad (Sorted Quads) uses about 2 *rpt*. The storage result is a 33% improvement over SOT and a 69% improvement over ECT. SQuad, which is discussed in Chapter 5, is an extension of SOT. SQuad utilizes the principles outlined in SOT, and also the principle of pairing to further reduce storage. In SQuad, most of the matched triangles are paired with an adjacent unmatched triangles. The quads they form are ordered to match the input order of the vertices.

- LR uses about 1 *rpt* (or equivalently 32 *bpt*), and its BELR (Bit-Efficient LR version) uses about 26 *bpt*. The storage result is a 50% (60% for BELR, assuming 32-bit references) improvement over SQuad and an 85% (87% for BELR) improvement over ECT. LR, discussed in Chapter 6, utilizes the principles of matching, pairing, reordering and chaining to reduce storage. In LR, quads (triangle pairs) are reordered so that their diagonals form a loop (which we call the ring) that visits most of the vertices (and hence is a nearly-Hamiltonian cycle). With most ring vertices, LR stores two references to the tip vertices of the matched quad. BELR stores these references as relative offsets to reduce storage.

- Zipper uses about 6 *bpt* (or equivalently 0.19 *rpt*). This corresponds to an 81% storage reduction over LR and a 97% reduction over ECT. Zipper, discussed in Chapter 7, is an extension of LR. Zipper utilizes the principles outlined in BELR, but uses a more effective formula for the relative offsets (deltas). Consecutive deltas are grouped into blocks of size 32. Zipper provides an efficient approach for computing the actual reference directly without having to iterate through the entire block.

Our methods have been designed to support triangle meshes of fixed topology. These limitations have been overcome by extensions of our work by various authors,

which are discussed in Section 9.1. For example, an extension of SOT to tetrahedral meshes has been proposed by Gurung and Rossignac [24]. Support of local connectivity changes (such as edge-flips, triangle-splits, or valence-3 vertex removals) has been addressed by Castelli Aleardi, Devillers and Rossignac in their ESQ (Editable Squad) [14] extension of SQuad. Our methods impose a new order for the triangles and/or vertices of the mesh, and thus make our data structures incompatible with streamed processing. To address this limitation, Luffel, Gurung, Lindstrom and Rossignac proposed Meshlets [30], which extends SQuad to make it streamable.

Our methods offer different tradeoffs between storage and performance. They may be less useful when processing small meshes that fit in core memory. However, for large meshes, the storage reduction helps reduce the frequency of page faults when accessing elements of a mesh that do not fit in memory, hence improving performance. This has been experimentally verified and is discussed in Chapter 8.

REFERENCES

[1] Attene, M., Falcidieno, B., Spagnuolo, M., and Rossignac, J., "Swingwrapper: Retiling triangle meshes for better edgebreaker compression," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 4, pp. 982–996, 2003.

[2] Autodesk, Inc., *Autodesk 3ds Max*, http://usa.autodesk.com/3ds-max/.

[3] Autodesk, Inc., *Autodesk AutoCAD*, http://usa.autodesk.com/autocad/.

[4] Autodesk, Inc., *Autodesk Maya*, http://usa.autodesk.com/maya/.

[5] Barbay, J., Castelli Aleardi, L., He, M., and Munro, J., "Succinct representation of labeled graphs," in *Algorithms and Computation* (Tokuyama, T., ed.), vol. 4835 of *Lecture Notes in Computer Science*, pp. 316–328, Springer Berlin / Heidelberg, 2007.

[6] Baumgart, B. G., "Winged edge polyhedron representation," Tech. Rep. CS-TR-72-320, Stanford University, 1972.

[7] Biedl, T. C., Bose, P., Demaine, E. D., and Lubiw, A., "Efficient algorithms for petersen's matching theorem," *J. Algorithms*, vol. 38, no. 1, pp. 110–134, 2001.

[8] Botsch, M., Kobbelt, L., Pauly, M., Alliez, P., and Lévy, B., *Polygon mesh processing*. AK Peters Limited, 2010.

[9] Brisson, E., "Representing geometric structures in d dimensions: Topology and order," in *ACM Symposium on Computational Geometry*, pp. 218–227, 1989.

[10] Campagna, S., Kobbelt, L., and Seidel, H.-P., "Directed edges—A scalable representation for triangle meshes," *Journal of Graphics Tools*, vol. 3, no. 4, pp. 1–11, 1998.

[11] Castelli Aleardi, L. and Devillers, O., "Catalog based representation of 2D triangulation," *Internat. J. Comput. Geom. Appl.*, vol. 21, pp. 393–402, 2011.

[12] Castelli Aleardi, L. and Devillers, O., "Explicit array-based compact data structures for triangulations," Tech. Rep. 00623762, INRIA, 2011.

[13] Castelli Aleardi, L., Devillers, O., and Mebarki, A., "2D triangulation representation using stable catalogs," in *Proc. 18th Canad. Conf. Comp. Geom*, pp. 71–74, 2006.

[14] Castelli Aleardi, L., Devillers, O., and Rossignac, J., "ESQ: Editable SQuad representation for triangle meshes," in *SIBGRAPI 2012 (XXV Conference on Graphics, Patterns and Images)* (C. Freitas, L. S., Scopigno, R., , and Sarkar, S., eds.), (Ouro Preto, MG, Brazil), august 2012.

[15] Castelli Aleardi, L., Devillers, O., and Schaeffer, G., "Succinct representations of planar maps," *Theoretical Computer Science*, vol. 408, no. 2–3, pp. 174–187, 2008.

[16] Chica, A., Williams, J., Andujar, C., Brunet, P., Navazo, I., Rossignac, J., and Vinacua, A., "Pressing: Smooth isosurfaces with flats from binary grids," *Computer Graphics Forum*, vol. 27, no. 1, pp. 36–46, 2008.

[17] Deering, M., "Geometry compression," in *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, (New York, NY, USA), pp. 13–20, ACM, 1995.

[18] Fotakis, D., Pagh, R., Sanders, P., and Spirakis, P., "Space efficient hash tables with worst case constant access time," *Lecture Notes in Computer Science*, vol. 2607, pp. 271–283, 2003.

[19] Gumhold, S., "New bounds on the encoding of planar triangulations," tech. rep., Universitt Tbingen, 2000.

[20] Gumhold, S. and Strasser, W., "Real time compression of triangle mesh connectivity," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, (New York, NY, USA), pp. 133–140, ACM, 1998.

[21] Gurung, T., Laney, D., Lindstrom, P., and Rossignac, J., "SQuad: Compact representation for triangle meshes," *Computer Graphics Forum*, vol. 30, no. 2, pp. 355–364, 2011.

[22] Gurung, T., Luffel, M., Lindstrom, P., and Rossignac, J., "LR: compact connectivity representation for triangle meshes," in *ACM SIGGRAPH 2011 papers*, SIGGRAPH '11, (New York, NY, USA), pp. 67:1–67:8, ACM, 2011.

[23] Gurung, T., Luffel, M., Lindstrom, P., and Rossignac, J., "Zipper: A compact connectivity data structure for triangle meshes," *Computer-Aided Design*, vol. 45, no. 2, pp. 262 – 269, 2013. Solid and Physical Modeling 2012.

[24] Gurung, T. and Rossignac, J., "SOT: compact representation for tetrahedral meshes," in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, (New York, NY, USA), pp. 79–88, ACM, 2009.

[25] Gurung, T. and Rossignac, J., "SOT: Compact representation for triangle and tetrahedral meshes," *Georgia Tech Technical Report*, 2010.

[26] Kallmann, M. and Thalmann, D., "Star-vertices: A compact representation for planar meshes with adjacency information," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 7–18, 2001.

[27] Karp, R. M., "Reducibility among Combinatorial Problems," *Complexity of Computer Computations*, 1972.

[28] King, D. and Rossignac, J., "Guaranteed 3.67v bit encoding of planar triangle graphs," in *11TH Canadian Conference on Computational Geometry (CCCG'99*, pp. 146–149, 1999.

[29] Lorensen, W. and Cline, H., "Marching cubes: A high resolution 3d surface construction algorithm," in *ACM Siggraph Computer Graphics*, vol. 21, pp. 163–169, ACM, 1987.

[30] Luffel, M., Gurung, T., Lindstrom, P., and Rossignac, J., "Meshlets: A compact, streamable triangle mesh data structure," *TVCG*, under review.

[31] Mantyla, M., *Introduction to Solid Modeling*. W. H. Freeman & Co., 1988.

[32] Petersen, J., "Die theorie der regulären graphs," *Acta Mathematica*, vol. 15, no. 1, pp. 193–200, 1891.

[33] Rossignac, J. and Szymczak, A., "Wrap&zip decompression of the connectivity of triangle meshes compressed with edgebreaker," *Journal of Computational Geometry*, vol. 14, no. 1-3, pp. 119–135, 1999.

[34] Rossignac, J., "Edgebreaker: Connectivity compression for triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 5, no. 1, pp. 47–61, 1999.

[35] Rossignac, J., "3D compression made simple: Edgebreaker with zip&wrap on a corner-table," in *International Conference on Shape Modeling & Applications*, pp. 278–283, 2001.

[36] Rossignac, J. and Cardoze, D., "Matchmaker: manifold breps for non-manifold r-sets," in *Proceedings of the fifth ACM symposium on Solid modeling and applications*, SMA '99, (New York, NY, USA), pp. 31–41, ACM, 1999.

[37] Rossignac, J. and Requicha, A., "Constant-radius blending in solid modeling," *ASME Computers In Mechanical Engineering (CIME)*, vol. 3, pp. 65–73, 1984.

[38] Rossignac, J., Safonova, A., and Szymczak, A., "Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces," in *Hierarchical and Geometrical Methods in Scientific Visualization*, pp. 41–50, Springer Verlag, 2003.

[39] Sieger, D. and Botsch, M., "Design, implementation, and evaluation of the surfacemesh data structure," *International Meshing Roundtable*, 2011.

[40] Snoeyink, J. and Speckmann, B., "Tripod: A minimalist data structure for embedded triangulations," in *Computational Graph Theory and Combinatorics*, 1999.

[41] Tarini, M., Pietroni, N., Cignoni, P., Panozzo, D., and Puppo, E., "Practical quad mesh simplification," *Computer Graphics Forum*, vol. 29, no. 2, pp. 407–418, 2010.

[42] Touma, C. and Gotsman, C., "Triangle mesh compression," *Proceedings Graphics Interface*, pp. 26–34, 1998.

[43] Tutte, W., "A census of planar triangulations," *Canadian J. of Mathematics*, vol. 14, pp. 21–38, 1962.

[44] Ueng, S.-K. and Sikorski, K., "A note on a linear time algorithm for constructing adjacency graphs of 3d fea data," 1996.

[45] Yang, X., *Three-Dimensional characterization of inherent and induced sand microstructure*. PhD thesis, Georgia Institute of Technology, 2005.

[46] Yoon, S.-E. and Lindstrom, P., "Random-accessible compressed triangle meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1536–1543, 2007.

VITA

Topraj Gurung grew up in the beautiful city of Pokhara, Nepal which is nestled in the Himalayas. He did his schooling at Gandaki Boarding School in Pokhara and completed his grade 11 and 12 (I.Sc.) at St. Xavier's Campus in the bustling city of Kathmandu, Nepal. He graduated with a Bachelor's degree from Georgia Institute of Technology in 2004 and with a Master's degree from Georgia Tech in 2007.

In 2007, Topraj enrolled in the Ph.D. program at Georgia Tech where he worked on data structures for triangle and tetrahedral meshes. He was advised by Professor Jarek Rossignac and Professor David Frost. During his Ph.D., he interned at Lawrence Livermore National Laboratory in Livermore, California in the summer of 2010 where he was hosted by Dr. Peter Lindstrom and Dr. Daniel Laney. During his internship, he worked on data structures for triangle meshes. He also interned at Medicsight LLC in London, UK in the summer of 2011 where he was hosted by Dr. Greg Slabaugh and Dr. Xujiong Ye. While there, he worked on efficient methods for mesh parameterization with application to non-rigid mesh registration.